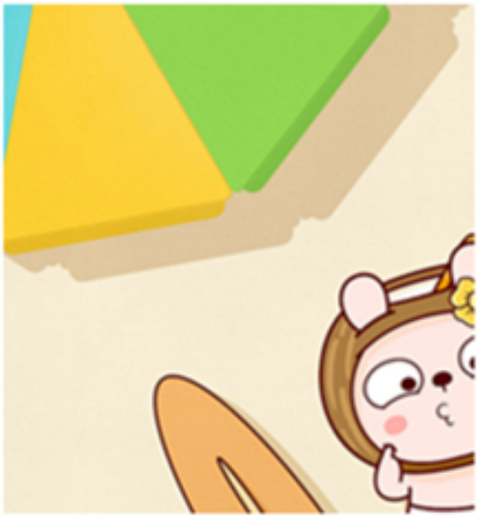
最新IT书籍PDF
最新IT书籍PDF
最新IT书籍PDF
最新IT书籍PDF



版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

微服务

架构实战

基于 Spring Boot、Spring Cloud、Docker

郑天民 / 著

结合作者自身在互联网行业实施服务化的经历讲解微服务技术体系与实践

Spring Boot、Spring Cloud和Docker等主流技术构建微服务架构

结合业务场景，通过全面案例给出工具在构建微服务架构中的工程实践

中国工信出版集团

人民邮电出版社
POSTS & TELECOM PRESS

仅供非商业用途或交流学习使用

微服务架构实战

基于 Spring Boot、Spring Cloud、Docker

郑天民／著

人民邮电出版社
北京

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

图书在版编目（CIP）数据

微服务架构实战 : 基于Spring Boot、Spring Cloud、Docker / 郑天民著. -- 北京 : 人民邮电出版社, 2018.12
ISBN 978-7-115-49090-2

Ⅰ. ①微… Ⅱ. ①郑… Ⅲ. ①互联网络－网络服务器 Ⅳ. ①TP368.5

中国版本图书馆CIP数据核字(2018)第185411号

内 容 提 要

本书主要介绍微服务架构实现过程中所应具备的技术体系。全书围绕实现微服务架构的基础组件和关键要素，讨论了使用 Spring Boot、Spring Cloud、Docker 等技术体系构建服务治理、负载均衡、服务容错、服务网关、配置中心、事件驱动、服务安全、服务监控、服务测试和服务部署等核心主题，并基于这些核心主题给出了具体的案例分析。

本书面向立志于成为微服务架构师的后端开发人员，读者不需要有很深的技术水平，也不限于特定的开发语言，但是熟悉 Java EE 常见技术并掌握系统设计基本概念，将有助于更好地理解书中的内容。同时，本书也可供具备不同技术体系的架构师同行参考。

◆ 著　　郑天民
责任编辑　刘　博
责任印制　彭志环
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
固安县铭成印刷有限公司印刷
◆ 开本：800×1000　1/16
印张：18.25　　2018 年 12 月第 1 版
字数：399 千字　　2018 年 12 月河北第 1 次印刷

定价：59.80 元

读者服务热线：(010)81055256　印装质量热线：(010)81055316
反盗版热线：(010)81055315

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

前言

当下互联网行业飞速发展，快速的业务更新和产品迭代给系统开发带来了新的挑战。围绕来自系统开发团队外部及内部的变化与挑战，如何更为合理地划分系统和团队边界，如何更加有效地组织系统开发过程，如何通过技术手段识别和消除开发过程中存在的浪费，成为广大系统开发人员和管理人员亟待解决的问题。在这种背景下，微服务架构应运而生了。要想实现微服务架构，不但要了解微服务的设计原理，还要掌握相应的技术体系。关于前者，笔者已出版了《微服务设计原理和架构》一书。本书主要面向的是后者，即围绕微服务架构的实现技术给出相应的工具框架和工程实践，旨在为广大开发人员提供对主流微服务架构实现技术的完整、全面和实用的介绍。

全书围绕实现微服务架构的基础组件和关键要素，引入 Spring Boot、Spring Cloud、Docker 等技术体系构建微服务体系。本书在组织结构上分为如下所示的 12 章，并结合这些章节内容要点给出一个完整的实现案例。

第 1 章　微服务架构设计，简要介绍微服务的建模方法、服务拆分和集成、基础组件和关键要素以及实现技术。

第 2 章　使用 Spring Boot 构建服务，引入 Spring Boot 作为单个微服务的实现框架。

第 3 章　Spring Cloud Netflix Eureka 与服务治理，引入 Eureka 实现服务注册和发现。

第 4 章　Spring Cloud Netflix Ribbon 与负载均衡，引入 Ribbon 实现客户端负载均衡。

第 5 章　Spring Cloud Netflix Hystrix 与服务容错，引入 Hystrix 实现面向消费者的服务容错。

第 6 章　Spring Cloud Netflix Zuul 与 API 网关，引入 Zuul 实现 API 网关。

第 7 章　Spring Cloud Config 与配置中心，引入 Spring Cloud Config 实现分布式配置中心。

第 8 章　Spring Cloud Stream 与事件驱动，引入 Spring Cloud Stream 实现服务之间的消息传递。

第 9 章　Spring Cloud Security 与服务安全，引入 Spring Cloud Security 实现服务的安全访问。

第 10 章　Spring Cloud Sleuth 与服务监控，引入 Spring Cloud Sleuth 实现服务的跟踪和监控。

第 11 章　Spring Test 与服务测试，引入 Spring Test 实现多层次的服务测试。

第 12 章　Docker 与服务部署，引入 Docker 实现服务的高效部署。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

通过对全书内容的系统学习，读者将对微服务架构的技术体系和实现机制有全面而深入的了解，为后续的工作和学习铺平道路。

在本书的撰写过程中，我的家人特别是我的妻子给予了我极大的支持和理解。感谢以往以及现在公司的同事们，身处业界领先的公司和团队，让我得到很多学习及成长的机会，没有平时大家的帮助，就不可能有本书的诞生！

本书所涉及的源代码，读者可到人邮教育社区（www.ryjiaoyu.com）下载。

由于时间仓促，编者水平有限，书中难免存在不足及疏漏之处，恳请广大读者批评指正。有任何建议和意见，可关注微信公众号“程序员向架构师转型”，与作者交流。

郑天民

2018 年 5 月

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

目录

第 1 章 微服务架构设计 1

1.1 直面微服务架构 1

1.1.1 分布式系统与微服务架构 1

1.1.2 微服务架构的优势与挑战 3

1.1.3 实施微服务架构 5

1.2 服务建模方法 6

1.2.1 服务的模型 6

1.2.2 服务的边界 7

1.2.3 服务的数据 7

1.3 服务拆分与集成 8

1.3.1 服务拆分 8

1.3.2 服务集成 9

1.4 微服务架构的基础组件和关键要素 10

1.4.1 微服务架构的基础组件 11

1.4.2 微服务架构的关键元素 11

1.5 实现微服务架构 13

1.5.1 微服务架构技术体系 13

1.5.2 微服务架构实现技术选型 14

1.6 案例分析 17

1.7 全书架构 18

1.8 本章小结 19

第 2 章 使用 Spring Boot 构建服务 21

2.1 引入 Spring Boot 21

2.2 基于 Spring Boot 的第一个服务 23

2.2.1 环境准备 23

2.2.2 实现 RESTful 服务 24

2.3 Spring Boot 常见功能 28

2.3.1 集成 Spring Data 28

2.3.2 集成消息中间件 37

2.3.3 系统监控 39

2.4 Spring Boot 基本原理 44

2.5 本章小结 46

第 3 章 Spring Cloud Netflix Eureka 与服务治理 47

3.1 服务治理解决方案 48

3.1.1 服务治理的需求和模型 48

3.1.2 服务治理的基本方案 49

3.2 构建 Eureka 服务 52

3.2.1 构建单个 Eureka 服务器 52

3.2.2 构建 Eureka 服务器集群 55

3.3 使用 Eureka 注册和发现服务 57

3.3.1 通过配置实现服务注册 58

3.3.2 获取服务注册信息 59

3.4 Eureka 基本架构 61

3.4.1 Eureka 服务注册和发现架构 61

3.4.2 Eureka 高可用架构 64

3.4.3 Eureka 区域亲和性 65

3.5 本章小结 66

第 4 章 Spring Cloud Netflix Ribbon 与负载均衡 67

4.1 负载均衡 68

4.1.1 负载均衡的类型 68

4.1.2 负载均衡的算法 70

4.2 使用 Ribbon 实现客户端负载均衡 71

4.2.1 Spring Cloud Netflix Ribbon 简介 71

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

4.2.2 使用 DiscoveryClient 查找服务 72
4.2.3 通过 RestTemplate 调用服务 74
4.3 Ribbon 基本架构 78
4.3.1 Ribbon 核心机制 79
4.3.2 Ribbon 负载均衡策略 81
4.3.3 @LoadBalanced 注解与 RestTemplate 82
4.3.4 @RibbonClient 注解与自定义负载均衡策略 83
4.4 本章小结 85
第 5 章 Spring Cloud Netflix Hystrix 与服务容错 86
5.1 服务消费者容错思想和模式 87
5.1.1 服务消费者容错的需求 87
5.1.2 服务隔离 88
5.1.3 服务熔断 90
5.1.4 服务回退 91
5.2 使用 Hystrix 实现服务容错 91
5.2.1 引入 Hystrix 92
5.2.2 使用 Hystrix 实现服务隔离 93
5.2.3 使用 Hystrix 实现服务熔断 96
5.2.4 使用 Hystrix 实现服务回退 99
5.3 Hystrix 基本原理 101
5.3.1 服务隔离 101
5.3.2 服务熔断 103
5.3.3 Hystrix 配置项 105
5.4 本章小结 109
第 6 章 Spring Cloud Netflix Zuul 与 API 网关 110
6.1 服务网关的设计理念 111
6.1.1 服务网关的作用 111
6.1.2 服务网关的结构和功能 112
6.2 使用 Zuul 构建服务网关 113
6.2.1 构建 Zuul 服务器 113
6.2.2 配置 Zuul 服务路由 115
6.3 Zuul 基本架构 120
6.3.1 ZuulFilter 组件架构 120
6.3.2 使用 Zuul 过滤器 124
6.4 本章小结 129
第 7 章 Spring Cloud Config 与配置中心 130
7.1 分布式配置中心方案 131
7.1.1 分布式配置模型 131
7.1.2 配置中心实现工具 133
7.2 构建配置中心服务器 134
7.2.1 引入 Spring Cloud Config 134
7.2.2 实现基于本地文件系统的配置方案 135
7.2.3 实现基于 Git 的配置方案 139
7.3 使用配置服务 140
7.3.1 访问配置项 141
7.3.2 配置数据安全性 145
7.4 Spring Cloud Config 特性 148
7.4.1 Spring Cloud Config 对比 Zookeeper 148
7.4.2 Spring Cloud Config 高可用 149
7.5 本章小结 150
第 8 章 Spring Cloud Stream 与事件驱动 151
8.1 事件驱动架构与模型 152
8.1.1 基本事件驱动架构与实现机制 152
8.1.2 事件驱动与领域模型 155
8.2 引入 Spring Cloud Stream 157
8.2.1 Spring Cloud Stream 基本架构 157

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

8.2.2 Spring Cloud Stream 与 Spring Integration 159
8.2.3 Spring Cloud Stream 与消息中间件 162
8.3 实现消息发布者 165
8.3.1 消息发送场景与实现流程 165
8.3.2 在服务中添加消息发布者 166
8.4 实现消息消费者 170
8.4.1 消息消费场景与实现流程 170
8.4.2 在服务中添加消息消费者 172
8.5 本章小结 177

第 9 章 Spring Cloud Security 与服务安全 178

9.1 服务访问安全性与 OAuth 协议 178
9.1.1 微服务架构中的安全性设计 179
9.1.2 OAuth 协议 180
9.2 构建 OAuth 认证服务器 184
9.2.1 引入 Spring Cloud Security 185
9.2.2 初始化用户与客户端 186
9.2.3 生成 Token 191
9.3 使用 OAuth 保护服务访问 195
9.3.1 集成 OAuth 认证服务 195
9.3.2 创建服务访问策略 196
9.3.3 使用 OAuth2RestTemplate 传播 Token 201
9.4 本章小结 206

第 10 章 Spring Cloud Sleuth 与服务监控 207

10.1 服务监控与 Spring Cloud Sleuth 207
10.1.1 服务监控基本原理 207
10.1.2 引入 Spring Cloud Sleuth 209
10.2 整合 Spring Cloud Sleuth 与 Zipkin 215
10.2.1 Zipkin 基本结构 215
10.2.2 引入 Zipkin 216
10.2.3 使用 Zipkin 跟踪服务调用链路 218
10.2.4 使用 Zipkin 实现自定义跟踪 226
10.3 本章小结 228

第 11 章 Spring Test 与服务测试 230

11.1 微服务测试的方法 231
11.1.1 单元测试 231
11.1.2 集成测试 233
11.1.3 端到端测试 233
11.2 测试 Spring Boot 应用程序 234
11.2.1 初始化测试环境 234
11.2.2 执行单元测试 237
11.3 使用 Mock 和注解实施集成测试 241
11.3.1 使用@JsonTest 注解测试 JSON 数据 242
11.3.2 使用@DataJpaTest 注解测试 Repository 层 244
11.3.3 使用 Mock 测试 Service 层 248
11.3.4 使用 Mock 和@WebMvcTest 注解测试 Controller 层 252
11.4 消费者驱动的契约测试 254
11.4.1 面向契约的端对端测试 254
11.4.2 实现面向契约的端对端测试 257
11.5 本章小结 266

第 12 章 Docker 与服务部署 267

12.1 Docker 与微服务架构 267
12.1.1 Docker 的优势 268
12.1.2 Docker 组件与命令 268

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

12.2 使用 Dockerfile 构建服务镜像 272
12.2.1 Dockerfile 命令 272
12.2.2 使用 Dockerfile 命令构建镜像 273
12.3 使用 Docker Compose 编排服务 276
12.3.1 Docker Compose 组件与命令 276
12.3.2 使用 Docker Compose 279
12.3.3 Docker Compose 案例分析 281
12.4 本章小结 283
参考文献 284

仅供非商业用途或交流学习使用

第 1 章

微服务架构设计

近年来，微服务架构（Microservices Architecture）已经成为一种主流的软件开发方法论，它把一种特定的软件应用设计方法描述为能够独立部署的服务套件。所谓微服务（Microservices），就是一些具有足够小的粒度、能够相互协作且自治的服务体系。每个微服务都比较简单，仅关注于完成一个功能并能很好地完成该功能，而这里的功能代表的是一种业务能力。构建微服务体系需要一套完整的方法论和工程实践，而微服务架构的提出代表的就是实现微服务体系的架构模式，即为我们提供了这些方法论和工程实践。从这个角度讲，微服务架构需要我们理解、学习并应用到日常开发过程中去。

微服务架构基于分布式系统，同时借助了面向服务架构和企业服务总线的设计理念并做了改进和优化，从而形成一种新的架构体系。微服务架构一方面具备技术、业务和组织上的优势，另一方面也在技术架构和研发过程中存在巨大挑战。微服务架构的实施需要具备一定的前提，而构建微服务架构是一项系统工程，涉及服务建模、实现技术、基础设施和研发过程等各个维度。在实施过程中，也需要根据现有系统的具体情况采用合适的实施模式。

本章作为全书的开篇，对微服务设计原理与架构做了全面介绍。本书的关注点是微服务架构的实现技术，本章也会梳理目前市面上主流的微服务技术体系并完成技术选型。在本章的最后，我们还会给出全书的组织架构。

1.1 直面微服务架构

顾名思义，微服务区别与其他服务体系的关键在于它的“微”特性。“微”是小的同义词，所以容易让人联想到微服务都是小型的服务，这是微服务的第一个特性。微服务之间只有通过相互的协作和交互才能构成完整的服务体系，而这种协作和交互机制也是微服务区别其他服务体系的另一个主要方面。

1.1.1 分布式系统与微服务架构

所谓分布式系统（Distributed System）是指硬件或软件组件分布在不同的网络计算机上，

彼此之间仅仅通过消息传递进行通信和协调的系统。从这个定义中可以看出，分布式系统包含两个区别于单块系统（Monolith System）的本质性特征：一个是网络，分布式系统的所有组件都位于网络之中，对于互联网应用而言，则位于更为复杂的互联网环境中；另一个是通信和协调，与单块系统不同，位于分布式系统中的各个组件只有通过约定、高效且可靠的通信机制进行相关协作，才能完成某一项业务功能。这些特征是我们在设计和实现分布式系统时首先需要考虑的。

分布式系统相较于单块系统在具备一定优势的同时，也存在一些我们不得不考虑的特性，包括但不限于网络传输的三态性、系统之间的异构性、数据一致性、服务的可用性等。以上问题是分布式系统的基本特性，我们无法避免，只能想办法进行利用和管理，这就给我们设计和实现分布式系统提出了挑战。微服务架构本质上也是一种分布式系统，但在遵循通用分布式特性的基础上，微服务架构还表现出一定的特殊特性。下面将围绕微服务架构的这些特殊特性展开讨论。

Martin Fowler 指出，微服务架构具有以下特点。

（1）服务组件化

所谓组件（Component）是一种可独立替换和升级的软件单元。在我们日常开发过程中，可能会设计和使用很多组件，这些组件可能服务于系统内部，也可能存在于系统所运行的进程之外。而服务就是一种进程外组件，服务之间利用诸如 RPC（Remote Procedure Call，远程过程调用）等通信机制完成交互。服务组件化的主要目的是服务可以独立部署。如果某个应用程序是由一个运行在独立进程中的很多组件组成，那么对任何一个组件的改变都将导致整个应用程序必须重新部署。但是如果把应用程序拆分成很多服务，通常情况下，只需要重新部署那个改变的服务即可。在微服务架构中，每个服务运行在其独立的进程中，服务与服务之间采用轻量级通信机制互相沟通。

（2）按业务能力组织服务

在寻找把一个大的应用程序进行拆分的方法时，研发过程一般都会围绕产品团队、UED 团队、APP 前端团队和服务器端团队而展开，这些团队也就是通常所说的职能团队（Function Team）。当使用这种模式对团队进行划分时，任何一个需求变更，无论大小，都将导致跨团队协作，从而增加沟通和协作成本。而微服务架构下的划分方法则有所不同，它倾向于围绕业务功能的组织来分割服务。这些服务面向的是具体业务结构，而不是面向某项技术能力。因此，团队是跨职能的（Cross-Functional）特征团队（Feature Team），包含用户体验、项目管理和技术研发等开发过程所要求的所有技能。每个服务都围绕着业务进行构建，并且能够被独立地部署到生产或类生产环境。

（3）去中心化

服务集中治理的一种好处是在单一平台上进行标准化，而采用微服务的团队更喜欢不同的标准。把集中式系统中的组件拆分成不同的服务，我们在构建这些服务时就会有更多的选择。

对具体的某一个服务而言，应该根据业务上下文，选择合适的语言和工具进行构建。

此外，微服务架构也崇尚于对数据进行分散管理。当集中式的应用使用单一逻辑数据库进行数据持久化时，通常选择在应用的范围内使用一个数据库。然后，微服务让每个服务管理自己的数据库，无论是相同数据库的不同实例，还是不同的数据库系统。

（4）基础设施自动化

许多使用微服务架构的产品或者系统，它们的团队拥有丰富的持续集成（Continuous Integration）和持续交付（Continuous Delivery）经验。团队使用微服务架构构建软件需要更广泛的依赖基础设施自动化技术。

在微服务中同样需要考虑服务容错性设计等分布式系统所需要考虑的问题，我们对以上特点进行总结和提炼，认为微服务具备业务独立、进程隔离、团队自主和交付独立性等“微”特性。

1.1.2 微服务架构的优势与挑战

微服务并不是一个纯技术的事物，而是涉及业务边界、基础设施、组织架构等的综合体系。微服务架构存在明显的优势，但也面临着各种挑战。

1. 微服务架构的优势

微服务架构能够为我们带来许多优势，包括技术上的优势、业务结构上的优势及组织上的优势。

（1）技术优势

微服务架构提供的是一种高内聚、低耦合的组件化方案。组件所能带来的独立性与健壮性微服务都可以具备，但是微服务的组件化特征更多的表现在对业务的提炼和对边界的思考。使用微服务架构迫使我们使用诸如领域驱动设计（Domain Driven Design，DDD）的思想去开展策略设计和技术设计，从而为更好地划分业务功能、提取界限上下文和开展系统集成工作提供依据。而这些方法和依据的背后恰恰是我们在架构设计过程中经常会碰到的问题。

当系统架构转变成一系列微服务之间的通信和集成时，我们就明白具体实现技术已经不是系统设计和开发的主要约束条件。因为在微服务架构中，各个微服务之间使用的是轻量级的通信机制。所谓轻量级是指这些通信机制与具体实现技术无关，不受限于某一个特定协议或交互媒介。每个微服务高度独立，可以采用适合自身开发团队和技术体系的工具及框架来实现某个微服务，从而为我们提供了宝贵的技术自由度。

在微服务架构中，通过服务拆分与集成，单个服务在保持通信方式不变的前提下，对其内部功能和技术的改变不会对外部依赖它的服务产生任何影响。结合可扩展性的概念，微服务架构无疑具备这方面的优势。高扩展性往往能够带来高可伸缩性，因为可以伸缩的前提是对系统进行合理地拆分。当我们明确系统的运行瓶颈，并把引起这些瓶颈的业务功能构建成独立的微

服务，就可以采用服务集群等手段有效加强服务的运行环境和状态。

微服务是可用于改造遗留系统（Legacy System）的强有力武器。面对遗留系统，一方面该系统的技术体系可能存在设计上的重大缺陷，另一方面则是因为代码量巨大且不容易修改。在代码层次与遗留系统进行直接集成是痛苦且具有挑战性的工作，但是遗留系统以提供接口的方式暴露某些功能入口仍然是一个相对容易实现的过程。一旦获取这些接口，微服务架构就能与之进行通信并完成功能整合。

持续交付通过简单、可重复的流程来确保软件发布过程的可靠性，微服务作为独立的可部署单元，非常适合使用持续交付，因为每一项服务都可以在不依赖于其他服务的条件下完成发布和部署。基于微服务架构，持续交付管道可以运行得更快，从而加速问题反馈，这是持续交付的主要目标之一。而持续交付的另一个目标是降低系统风险，微服务小而独立，一旦出现问题很容易进行修复和回滚操作。

（2）业务和组织优势

微服务架构的技术特征决定了开发各自微服务的团队之间只需要进行较少的协调工作，这为降低研发过程浪费提供了基础。从组织管理的角度出发，自治式团队较之集中式管理模式下的团队在团队建立和发展上能够得到更好的可扩展性。集中式团队普遍受限于技术上的约束和决策，诸如可用性、性能等非功能性需求都不是由某个业务功能和系统的独立团队所负责，而是需要集中式的、系统级别的实现和管理。微服务架构可以把大型团队打散成小型团队，小型团队比大型团队具有更低的失败可能性。从这点上讲，微服务架构还能够降低团队组织级别的风险。

微服务架构从技术角度给出了缩短产品开发周期的方法，主要表现在并行的开发模式上。将业务拆分成多个微服务能够让不同的业务功能处于一种并行开发的状态，因为每个团队所负责的业务需求只影响到团队自身的微服务，所以各个团队能够独立开发，整个系统也很容易分布到各个团队中。如果涉及简单业务需求的变更或者是发布部署的要求，独立的微服务之间也不需要太多的统一协调工作。大规模的统一协调工作通常只发生在业务结构产生重大变更的场景，而这种场景对于软件开发而言显然是应该尽量避免并进行提前规划。

2. 微服务架构面临的挑战

将一个系统分散到多个微服务中使得系统整体结构变得更加复杂，这在技术架构和研发过程中同样给我们带来了一定挑战。

（1）技术架构挑战

微服务架构所带来的独立性可能导致每个服务采用不同的技术体系。去中心化的设计思想意味着微服务之间不需要共享技术，然而缺少通用的技术体系同样也会加剧系统开发的复杂度。去中心化是微服务架构的基本特征之一，但当每个微服务团队都采用自身团队擅长的技术进行微服务构建时，导致的问题就是没有一个人能够明白系统中所采用的所有技术。尽管多数情况

下，我们并不需要深入到其他团队的微服务中，但从统一发布和运维等角度去看待整个系统时，这种技术复杂性就可能会是一个问题。我们需要把控中心化和去中心化之间的平衡性。

每个微服务都应该具备与其他服务保持相互独立的版本控制机制，我们提倡为每个微服务建立版本并根据业务迭代更新这个版本。如果每个服务都能够完全按照自身服务的演进过程进行独立发布，那么事情还比较简单。但如果某一个服务与另一个服务具有版本关联性，即这两个服务要不都不发布，要不一起发布，那么事情就变得复杂了。如果这些具有版本关联性的服务很多且版本的更新频率很高，如何正确管理服务版本就是一项具有挑战性的工作。

（2）研发过程挑战

对研发过程而言，分散的服务在一定程度上等同于分散的业务需求，来源于微服务具有的业务独立性和明确的服务边界。但对于整个系统而言，需求的边界并不那么容易划分。当我们面对微服务架构，如何确定业务功能的粒度，如何把非功能性需求分解到各个微服务中，如何从系统整体上把握需求的优先级等都是我们不得不面对的问题。

对于大多数尚未采用微服务架构的团队而言，使用微服务架构就是一个引入变化的过程。关于这一点，很多人都存在一定的误解，最大的误解就是认为新想法一旦引入就意味着已经成功了。当我们尝试采用微服务架构时，团队管理人员想了很多办法、做了很多工作找到了问题的切入点，跟各个利益方讨论之后终于引入了大家都赞同的变化，然后就期望这个变化能按照自己想的那样发挥效果，这是不对的。当微服务架构被引入后，我们还要做很多事情，前面所提到的各种技术、架构和过程的挑战都需要我们进行跟踪和协调。

1.1.3 实施微服务架构

实施微服务架构存在固定的模式，最典型的实施模式是从一个单块系统逐步转变为多个维度的微服务架构。通常，不同的业务功能根据优先级、工作量等可以转化为一个一个的微服务。促使这种转变的主要目的可能来自于业务的快速变化和迭代要求，或者来自于将系统的部署与运维变得独立而简单，而更为重要的原因则可能来自于对架构可扩展性和可伸缩性的考虑。当然，从单块系统转变为微服务架构在方式上不同团队、不同场景可能也存在差异性，对于那些对服务的可用性要求很高的场景而言，可以先通过使用一些服务隔离手段设法提高服务的整体可靠性之后再拆分微服务；而有些场景下，对业务的分离可能是系统的最大痛点，那么快速、合理的拆分业务就是实施微服务架构的第一步。

当然，现实中也存在从无到有实施微服务架构的机会，尽管这种机会非常少见。即使有这种机会，一般的做法也是先构建一个单块系统，然后再设法加以改进，因为从无到有的过程就是一个从小到大的过程，而从微服务实施的前提上讲，小系统没有必要直接采用微服务架构。但在系统架构的初始设计阶段，采用领域驱动的设计思想建立粗粒度的领域模型是一种最佳实践，有助于后续向微服务架构转型的实施过程。领域驱动设计同样也驱使技术团队在设计阶段

就考虑到合适的团队分工和各个团队之间的独立性。

混合式是尝试微服务架构的第三种方式，因为微服务架构很容易与现有系统并存，这些微服务能够很好的对系统起到补充作用。如果需要，我们能够很快实现很多微服务，而一旦出现问题，移除这些微服务的成本也较低。微服务与遗留系统之间的易整合性是我们采用微服务的一个主要原因，关于这种整合关系及其实施方法可参考笔者所著的《微服务设计原理与架构》一书中的内容。

1.2 服务建模方法

如同系统架构设计首先需要进行业务建模一样，微服务架构的设计同样从服务建模开始。一般而言，软件行业建模工作的目的在于将业务需求转变为一种能够直接用于软件实现的载体，以 UML（Unified Modeling Language，统一建模语言）为代表的建模语言是这种载体的具体表现形式。在微服务架构中，服务建模的目的同样是为了便于开发人员开展后续的服务设计和实现工作。本节讨论微服务建模的切入点，包括服务的模型、服务的边界和服务的数据。

1.2.1 服务的模型

服务模型提供服务的概念模型，并给出服务的统一表现形式。服务的概念模型来自两个维度：一个是服务标准，即什么样的服务才是一个好的服务；另一个是服务级别，即不同的服务应该具备不同的重要性。

关于服务标准，SOA 体系中的很多设计理念同样适合于微服务架构，包括服务无状态（Service Statelessness）、服务可重用（Service Reusability）、服务可发现（Service Discoverability）、服务自治（Service Autonomy）和服务松耦合（Service Loose Coupling）等在内的设计原则都是服务设计上的核心标准。

关于服务级别，可以从发生具体事故时服务对用户体验的影响、所造成的经济损失等角度对服务进行具体分级。在服务分级上，一种常见的分级方法是将系统服务分成 3 个不同的等级：一级服务具备完善的容错降级机制及对低级别服务的熔断措施、定期压测、配置高级别的监控报警流程等；二级服务多采用异步方式进行系统交互，容忍暂时数据不一致性；三级服务则可随时降级整个服务。

服务需要具备统一的表现形式，也就是需要具备契约化的约束条件，而这种契约化的约束条件一般可以通过文档的方式进行展现。对于每一个服务而言都应该提供一份契约文档，并发布到统一的管理平台以方便服务相关人员进行查看和更新。服务契约（Service Contract）化要求至少对服务的几个基本方面作出说明，包括具体接口的 API 接入技术说明；服务能力的描述

（如服务所属的业务模块、所能提供的业务功能以及具体的应用场景）；提供这些能力所约定的一些限制条件说明（这些限制条件多数与业务场景有关，现实中以下这种情况并不少见，即同一个接口定义以及其所代表的业务能力在不同的场景中会出现不同的表现形式）；支持的最新版本和历史版本的说明（这点对于微服务而言是必备的要素）。显然，将服务契约化有助于在开发人员之间形成统一语言，减少多余且可能反复的口头沟通，降低协作成本。

1.2.2 服务的边界

在明确了服务分类和服务模型之后，接下去的工作就是识别服务，识别服务的切入点在于识别服务与服务之间的边界（Boundary）。明确服务边界是下一节中将要介绍的服务拆分与集成的前提。

在微服务架构中，识别服务边界的方法可以参考领域驱动设计思想。在领域驱动设计中，有两个主要的设计维度，即设计的策略维度和设计的技术维度。其中，设计的策略维度更关注如何设计领域模型以及对领域模型的划分，其目的在于清楚划分不同的系统与业务关注点。策略维度是一个面向业务、具备较高层次的设计维度，更偏重于业务架构的梳理以及考虑如何把业务架构和技术架构相结合的问题。策略维度的通用语言（Ubiquitous Language）、子域（Sub-Domain）、界限上下文（Boundary Context）等概念为识别服务边界提供了一整套方法论。设计的技术维度包含聚合（Aggregate）、领域事件（Domain Event）等组件，有助于组织服务内部以及服务与服务之间进行交互的方式。关于领域驱动设计的更多内容可以参考相关资料。

在通过使用领域和界限上下文进行服务边界划分的过程中，也存在一些服务边界划分的原则。常见的边界划分原则包括服务关联度原则、业务能力职责单一原则、读写分离原则以及组织关系原则。

1.2.3 服务的数据

通过确定服务边界，服务在逻辑上就变成了一个个独立的个体，但是我们还要小心一个并不容易进行独立处理的因素，那就是数据。绝大多数服务都会依赖数据，而很多数据可能也会被一批服务所依赖。

规范化数据模型是传统关系型数据库设计的核心，即通过三大范式实现数据的有效存储，并为开发人员提供一整套对数据的操作方式。规范化数据模型有利有弊，它虽然为如何管理关系型数据提供了最佳设计理念，但同时也限制了数据查询的灵活性和高效性，特别是当查询涉及很多关联（Join）场景时，会导致查询性能严重低下。规范化数据模型的另一个问题在于中心化思想，即把数据统一存储在一个中央数据库中。当大量数据存在于同一数据库时容易造成

数据库访问瓶颈，从而影响数据访问性能，并为系统可用性埋下隐患。

针对规范化数据模型存在的以上问题，微服务架构中数据管理的基本思路就是数据去中心化。数据去中心化场景包括跨表查询场景、跨库查询场景以及技术耦合场景。在对数据去中心化进行不断尝试的过程中，采用“代码分离→重复数据库模式→迁移数据读写操作→抽取服务化接口”等步骤构成了完整的数据去中心化流程。关于各种数据去中心化场景、应对方式以及去中心化工作流程的详细描述，可以参考笔者所著的《微服务设计原理与架构》一书。

1.3 服务拆分与集成

本节将在服务建模的基础上，简要分析服务拆分的策略和手段，同时给出了对拆分之后的服务进行集成的各种实现方法和技术体系。

1.3.1 服务拆分

在微服务架构中，我们认为服务是业务能力的代表，需要围绕业务进行组织。服务拆分的关键在于正确理解业务，识别单体内部的业务领域及其边界，并按边界进行拆分。所以，微服务的拆分模式本质上是基于不同的业务进行拆分。

服务拆分存在两大维度，即业务与数据。业务体现在各种功能代码中，通过确定业务的边界，并使用领域与界限上下文、领域事件等技术手段可以实现拆分。而数据的拆分则体现在如何将集中式的中心化数据转变为各个微服务所拥有的独立数据，这部分工作同样十分具有挑战性。

关于业务和数据谁应该先拆分的问题，可以是先数据库后业务代码，也可以是先业务代码后数据库。在拆分中遇到的最大挑战可能是数据层的拆分，因为在数据库中，可能存在各种跨表连接查询、跨库连接查询，以及不同业务模块的代码与数据耦合得非常紧密的场景，这会导致服务的拆分变得困难。因此，在拆分步骤上推荐数据库先行。数据模型能否彻底分开，很大程度上决定了微服务的边界功能是否彻底划清。

根据系统自身的特点和运行状态，服务拆分的方法通常分为绞杀者与修缮者两种模式。绞杀者模式（Strangler Pattern）指的是在现有系统外围将新功能用新的方式构建为新的服务的策略，通过将新功能做成微服务方式，而不是直接修改原有系统，逐步地实现对老系统的替换。采用这种策略，随着时间的推移，新的服务就会逐渐“绞杀”老的系统。对于那些规模很大而又难以对现有架构进行修改的遗留系统，推荐采用绞杀者模式。而修缮者模式就像修房或修路一样，将老旧待修缮的部分进行隔离，并用新的方式对其进行单独修复。修复的同时，需保证与其他部分仍能协同功能。从这种思路出发，修缮者模式更多地表现为一种重构技术，具体实现可以参考 Martine Fowler 的 Branch By Abstraction 重构方法。

数据对于微服务架构而言同样可以认为是一种依赖关系，因为任何业务都需要使用某个数据容器作为持久化的机制或者数据处理的媒介，这里的数据容器不仅仅是指关系型数据库，而是泛指包括消息队列、搜索引擎索引以及各种 Nosql 在内的数据媒介。微服务架构中存在一种说法，即我们需要将微服务用到的所有资源全部嵌入该服务中，从而确保微服务的独立性。

1.3.2 服务集成

服务之间势必需要集成，而这种集成关系远比简单的 API 调用要复杂。对于微服务架构而言，我们的思路是尽量采用标准化的数据结构并降低系统集成的耦合度。微服务架构中服务之间的集成模式可以分为以下 4 大类，同时还可引入其他一些手段来促使服务与服务之间的有效集成。

（1）接口集成

接口集成是服务之间集成的常见手段，通常是基于业务逻辑的需要进行集成。RPC、REST、消息传递和服务总线都可以归为这种集成方式。

RPC（Remote Process Call，远程过程调用）架构是服务之间进行集成的最基本方式。在 RPC 架构实现思路上，远程服务提供者以某种形式提供服务调用的相关信息，远程代理对象通过动态代理拦截机制生成远程服务的本地代理，让远程调用在使用上就如同本地调用一样。而网络通信应该与具体协议无关，通过序列化和反序列化方式对网络传输数据进行有效传输。

REST（Representational State Transfer）从技术上讲也可以认为是 RPC 架构的一种具体表现形式，因为 RPC 架构中最基本的网络通信、序列化/反序列化、传输协议和服务调用等组件都能在 REST 中有所体现。但 REST 代表的并不是一种技术，也不是一种标准和规范，而是一种设计风格。要理解 RESTful 架构，最好的方法就是去理解它的全称 Representational State Transfer，这个词组直译过来就是“表现层状态转移”，其实它省略了主语，“表现层”其实指的是“资源”的“表现层”。所以，通俗来讲 REST 就是指资源在网络中以某种表现形式进行状态转移。

消息传递（Messaging）机制能够降低技术、空间和时间耦合。消息传递机制在消息发送方和消息接收方之间添加了存储转发（Store and Forward）功能。存储转发是计算机网络领域使用最为广泛的技术之一，其基本思想是先将数据缓存起来，再根据目的地址将该数据发送出去。显然，有了存储转发机制之后，消息发送方与消息接收方之间并不需要相互认识，也不需要同时在线，更加不需要采用同样的实现技术。紧耦合的单阶段方法调用就转变成松耦合的两阶段过程，技术、空间和时间上的约束通过中间层得到显著缓解，这个中间层就是消息传递系统（Messaging System）。

企业服务总线（Enterprise Service Bus，ESB）本质上是一种系统集成组件，用于解决分布式环境下的异步协作问题，可以看作是对消息传递系统的扩展和延伸。ESB 提供了一批核心组件，包括路由器、转换器和端点。路由器（Router）在一个位置上维护消息目标地址并基于

消息本身或上下文进行路由；转换器（Transformer）用于异构系统之间进行数据适配，数据结构、类型、表现形式、传输方式都是潜在的需要转换的对象；端点（Endpoint）封装了应用系统与服务总线系统的交互。

（2）数据集成

数据集成同样可以用于微服务之间的交互，共享数据库（Shared Database）是一种选择，还可以通过数据复制（Data Replication）的方式实现数据集成。

在微服务架构中，我们追求数据的独立性。但对于一些遗留系统而言，我们无法重新打造数据体系，数据复制就成为一种折中的集成方法。所谓数据复制，就是在不同的数据容器中保存同一份业务数据。这里的同一份业务数据的概念不在于数据内容的完全一致性，而是在于这些数据背后的业务逻辑的一致性。

（3）客户端集成

由于微服务是一个能够独立运行的整体，有些微服务会包含一些 UI 界面，这就意味着微服务之间也可以通过 UI 界面进行集成。站在某一个微服务的角度讲，调用它的服务就是该服务的客户端。关于客户端与微服务之间的集成可以分为以下 3 种方式，即直接集成、使用 FrontEnd 服务器和使用 API 网关。

直接集成的方式比较简单，就是客户端通过微服务提供的访问入口直接对微服务进行集成。这种方式适合于微服务数量不是太多的场景。如果采用直接集成的方式，服务按照业务模块进行边界划分和命名是一项最佳实践。

FrontEnd 服务器有时候也可以认为是一种 Portal（门户）机制，即把客户端所需要的 CSS、JavaScript 等公共资源统一放在 FrontEnd 服务器，然后每个微服务包含自身特有的 HTML 等客户端代码片段以及业务逻辑，通过集成 FrontEnd 服务器上的公共资源完成独立服务的运行。

当微服务数量较多且客户端集成场景比较复杂时，通常就需要单独抽取一层作为客户端访问的统一入口，这一层在微服务架构里有个专门的叫法称之为 API 网关（Gateway）。API 网关的主要作用是对后端的各个微服务进行整合，从而为不同的客户端提供定制化的内容。API 网关是微服务架构的基础组件，下一节还会再次介绍。

（4）外部集成

这里把外部集成单独剥离出来的原因在于现实中很多服务之间的集成需求来自外部服务的依赖和整合，在集成方式上也可以综合采用接口集成、数据集成和客户端集成。

1.4 微服务架构的基础组件和关键要素

微服务架构的实现技术可以分为两类：一类是基础组件；另一类是关键要素。本节将对这两类组件做简要介绍。

1.4.1 微服务架构的基础组件

微服务架构的实现首先需要提供一系列基础组件，既包括事件驱动架构、集群与负载均衡、服务路由等分布式环境下的通用组件，又包括 API 网关和配置管理等微服务架构所特有的功能组件。

事件驱动架构（Event-Driven Architecture，EDA）定义了一个设计和实现应用系统的架构风格，在这个架构风格里事件可传输于松散耦合的组件和服务之间。事件处理架构的优势在于当系统中需要添加另一个业务逻辑来完成整个流程时，只需要对该事件添加一个订阅者即可，不需要对原有系统做大量修改。考虑到在微服务架构中服务数量较多且需要对服务进行重构，事件处理架构在系统扩展性上的优势尤为明显。而在技术实现上，通过消息传递机制，我们不必花费太大代价就能实现事件驱动架构。

集群（Cluster）指的是将几台服务器集中在一起，实现同一业务。而负载均衡（Load Balance）就是将请求分摊到位于集群中的多个服务器上进行执行。负载均衡根据服务器地址列表所存放的位置可以分成两类，一类是服务器端负载均衡，一类是客户端负载均衡。以各种负载均衡算法为基础的分发策略决定了负载均衡的效果。

在集群化环境中，当客户端请求到达集群，如何确定由某一台服务器进行请求响应就是服务路由（Routing）问题。从这个角度讲，负载均衡也是一种路由方案，但是负载均衡的出发点是提供服务分发而不是解决路由问题，常见的静态、动态负载均衡算法也无法实现精细化的路由管理。服务路由的管理可以归为几个大类，包括直接路由、间接路由和路由规则。

API 网关本质上是一种外观模式（Facade Pattren）的具体实现，它是一种服务器端应用程序，并且是系统访问的唯一入口。API 网关封装了系统内部架构，为每个客户端提供一个定制的 API。同时，它可能还具有身份验证、监控、缓存、请求管理、静态响应处理等功能。在微服务架构中，API 网关的核心要点是所有的客户端和消费端都通过统一的网关接入微服务，在网关层处理通用的非业务功能。

在微服务架构中，一般都需要引入配置中心（Configuration Center）的相关工具。采用配置中心也就意味着采用集中式配置管理的设计思想。对于集中式配置中心而言，开发、测试和生产等不同的环境配置信息保存在统一存储媒介中，这是一个维度。而另一个维度就是分布式集群环境，需要确保集群中同一类服务的所有服务器保存同一份配置文件并且能够同步更新。

1.4.2 微服务架构的关键元素

在微服务架构中，还有一些实现上的关键要素需要考虑。基于服务注册中心的服务发布和订阅机制是微服务体系下实现服务治理的基本手段。同时，在保证服务的可靠性上，也需要考

虑服务容错、服务隔离、服务限流和服务降级等需求和实现方案。

1. 服务注册中心

在微服务架构中，服务治理（Service Governance）可以说是最为关键的一个要素，因为各个微服务都需要通过服务治理实现自动化的注册（Registration）和发现（Discovery）。服务治理的需求来自服务的数量。在服务数量并不是太多的场景下，服务消费者获取服务提供者地址的基本思路是通过配置中心，当服务的消费者需要调用某个服务时，基于配置中心中存储的目标服务的具体地址构建链路完成调用。这是常见的路由方案，但并不是一种好的方案。为了实现微服务架构中的服务注册和发现，通常都需要构建一个独立的媒介来管理服务的实例，这个媒介一般被称为服务注册中心（Service Registration Center）。

注册中心应该具备发布-订阅功能，即服务提供者可以根据服务的元数据发布服务，服务消费者可以通过对自己感兴趣的服务进行订阅获取包括服务地址在内的各项元数据。发布-订阅的功能还体现在数据变更推送上，即当注册中心服务定义发生变化时，主动推送变更到该服务的消费者从而实现间接路由。由于服务提供者和服务消费者同时依赖于注册中心，这就需要确保数据一致性，即在任何时候服务提供者和消费者都应该看到同一份数据。作为发布-订阅模式的一种实现，注册中心的这些功能都是与该模式的基本原理保持一致的。

为了确保服务高可用性，一般也需要注册中心本身保持高可用性，也就意味着注册中心本身需要构建对等集群。所谓对等集群（Peer-to-peer Server Cluster），是指集群中所有的服务器都提供同样的服务，所以在对等集群中，客户端只要连接一个服务器完成服务注册和发现即可，任何一台服务器宕机都不影响客户端正常使用。

作为注册中心的客户端程序，一般会嵌入在服务提供者和服务消费者的应用程序中。在应用程序运行时，服务提供者的注册中心客户端程序会向注册中心注册自身提供的服务，而服务消费者的注册中心客户端程序则从注册中心查询当前订阅的服务信息并周期性地刷新服务状态。同时，为了提高服务路由的效率和容错性，服务消费者可以配备缓存机制以加速服务路由，更重要的是当服务注册中心不可用时，服务消费者可以利用本地缓存路由实现对现有服务的可靠调用。

2. 服务可靠性方案

在微服务架构中，各个服务独立部署且服务与服务之间存在相互依赖的关系。和单块系统相比，微服务架构中出现服务访问失败的原因和场景非常复杂，这需要从服务可靠性的角度出发对服务自身以及服务与服务之间的交互过程进行分析。

对于服务提供者而言，一旦自身服务发生错误，那么应该快速返回合理的处理结果；而对于服务消费者而言，则重点关注不要被服务提供者所产生的错误影响到自身服务的可用性。

对于服务的失败，常见的应对策略有超时（Timeout）机制和重试（Retry）机制两种。超

时机制指的是调用服务的操作可以配置为执行超时，如果服务未能在这个时间内响应，将回复一个失败消息。同时，为了降低网络瞬态异常所造成的网络通信问题，可以使用重试机制。这两种方式都会产生同步等待，因此合理限制超时时间和重试次数是一般的做法。

当服务运行在一个集群中，出现通信链路故障、服务端超时以及业务异常等场景都会导致服务调用失败。容错（Fault Tolerance）机制的基本思想是冗余和重试，即当一个服务器出现问题时不妨试试其他服务器。集群的建立已经满足冗余的条件，而围绕如何进行重试就产生了 Failover、Failback 等几种常见的集群容错策略。

服务隔离（Isolation）包括一些常见的隔离思路以及特定的隔离实现技术框架。所谓隔离本质上是对系统或资源进行分割，从而实现当系统发生故障时能限定传播范围和影响范围，即发生故障后只有出问题的服务不可用，保证其他服务仍然可用。常见的隔离措施包括线程隔离、进程隔离、集群隔离、机房隔离和读写隔离等。

关于服务可靠性还有一个重要的概念——服务熔断（Circuit Breaker），服务熔断类似现实世界中的“保险丝”，当某个异常条件被触发就直接熔断整个服务，并不是一直等到此服务超时。而服务降级就是当某个服务熔断之后，服务端准备一个本地的回退（Fallback）方法，该方法返回一个默认值。

1.5 实现微服务架构

本书的定位是讨论实现微服务架构的技术体系和工程实践。在前面各节内容中，我们已经对服务建模、服务拆分、服务集成进行了分析，并给出了构建微服务架构的基础组件和关键要素，这些内容体现了实现微服务架构的需求。接下来我们将围绕如何实现这些需求展开讨论，通过分析目前市面上主流的技术体系完成微服务架构实现技术选型。

1.5.1 微服务架构技术体系

本书将从两个维度对微服务架构技术体系进行描述，即实现技术维度和管理体系维度。

为了实现微服务，首先需要选择一种主流的工具来构建单个微服务。当系统中存在多个微服务时，我们就应该提供服务治理、负载均衡、服务容错、API 网关、配置中心、事件驱动等实现方案以完成服务之间的交互和集成。

微服务架构管理体系包括如何对微服务进行测试，以及基于日志聚合和服务跟踪的服务监控管理。同时，服务安全对于微服务架构而言也提供了不同的实现策略和技术。最后，我们把服务的部署也归到管理体系之中。

1.5.2 微服务架构实现技术选型

实现微服务架构的第一步是进行技术选型，也就是选择一个合适的技术体系来支持微服务的开发工作。目前市面上并没有一个真正意义上实现微服务的技术体系，但还是存在一些可供参考的工具和框架。本书后续内容将使用 Spring Cloud 作为实现微服务的主体框架，本节将会给出选择该框架的原因，同时也会对其他同类框架进行分析和类比。

1. 技术选型的参考标准

所谓技术选型，首先需要明确选型的参考标准。结合前面的讨论，我们对于工具的选择也将围绕核心组件完备性和关键要素实现难度两方面来展开。

（1）核心组件完备性

微服务的核心组件由服务通信、事件驱动、负载均衡、服务路由、API 网关和配置管理所组成。

- 服务通信

对于服务通信，微服务架构明确要求服务之间通过跨进程的远程调用方式进行通信。关于远程调用，有 3 种风格的解决方案，即 RPC、REST 和自定义实现。而在服务与服务的交互方式上，也存在两个维度，即按照交互对象的数量分为一对一和一对多，以及按照请求响应的方式分为同步和异步。目前 RPC 框架可供选型的余地很大，如 Alibaba Dubbo、Facebook Thrift 及 Google gRPC 等都是非常主流的实现，而 REST 的实现则包括 Jersey 和 Spring MVC 等。

- 事件驱动

事件驱动架构实现工具的表现形式通常是各种消息中间件，如实现 JMS（Java Message Service，Java 消息服务）规范的 ActiveMQ、实现 AMQP（Advanced Message Queuing Protocol，高级消息队列协议）规范的 RabbitMQ、在大数据流式计算领域中应用非常广泛的 Kafka，当然还有像 Alibaba 自研的 RocketMQ。在这些消息中间件中，ActiveMQ 一般不考虑，如果是相对比较轻量级的应用可以选择 RabbitMQ，而 Kafka 则适合大型应用的场景。

- 负载均衡

负载均衡分为服务器端负载均衡和客户端负载均衡两大类实现方案。在服务器软件上，可以选择 Nginx、HA proxy、Apache、LVS 等工具。而 Netflix Ribbon 则是一种可以单独使用的负载均衡机制。事实上所有的分布式服务框架基本都内置了负载均衡的实现，所以负载均衡本身并不需要做太多的选择。

- API 网关

API 网关是微服务架构的核心组件之一。Netflix OSS（Open Source Software）中有一个 Zuul 提供了一套过滤器机制，可以很好地支持签名校验、登录校验等前置过滤功能处理，同时

它也维护了路由规则和服务实例，以便完成服务路由功能。其他可供参考的 API 网关还有 Mashape 和开源的 Kong 等。

- 配置管理

配置管理的作用是完成集中式的配置信息管理。目前比较主流的包括 Spring 旗下的 Spring Cloud Config、淘宝的 Diamond 和百度的 Disconf 等。在实现上，Spring Cloud Config 支持配置服务放在配置服务的内存中（即本地），也支持放在远程 Git 仓库，这点与其他工具在设计上有较大不同。Diamond 和 Disconf 都是基于 MySQL 作为存储媒介，Diamond 采用拉模型，即每隔 15 秒拉一次全量数据；而 Disconf 基于 Zookeeper 的推模型，实时推送。在配置数据模型上，Diamond 只支持 Key-Value 结构的数据，采用的是非配置文件模式；而 Disconf 支持传统的配置文件模式，也支持 Key-Value 结构数据。

（2）关键要素实现难度

关于微服务实现的关键要素，因为数据一致性更多是一种实现机制而不是具体工具，所以我们主要需要明确注册中心以及服务可靠性实现上的难度。

- 注册中心

关于服务注册和服务发现，比较常见的分布式一致性协议是 Paxos 协议和 Raft 协议。相比 Paxos 协议，Raft 协议易于理解和实现，因此最新的分布式一致性方案大都选择 Raft 协议。我们知道 Zookeeper 采用的是基于 Paxos 协议改进的 ZAB（Zookeeper Atomic Broadcast，Zookeeper 原子消息广播）协议，而 Raft 协议的实现工具主要是 Consul 和 Etcd。

注册中心是任何一个微服务框架所必不可少的组件，很多框架都内建了对服务注册中心的支持。例如，Alibaba 的 Dubbo 框架支持包括 Zookeeper、Redis 等在内的多种注册中心实现，默认采用的是 Zookeeper；新浪的 Motan 支持 Zookeeper，也支持 Consul；Spring Cloud 同时提供了 Spring Cloud Consul 和 Spring Cloud Zookeeper 两种实现方案；而 Netflix OSS 中使用的是 Eureka。

- 服务可靠性

服务可靠性相关的功能主要包括服务容错、服务隔离、服务限流和服务降级，其中大多数机制都偏向与实现策略而不是实现工具。我们需要明确的是如何实现服务隔离和服务熔断机制的框架。

服务熔断器目前只有一个可选的开源方案，那就是 Netflix Hystrix。Netflix Hystrix 同时也实现了服务隔离机制。本书后续章节会对该框架做详细分析。

2. 微服务实现框架对比

讲完微服务框架的选型标准以及所应该具备的核心组件和关键要素之后，下面来看一下目前业界有哪些现成的框架可供我们直接使用，这里列举了 Dubbo 和 Spring Cloud 这两个目前最主流的框架作为选型的基础。

（1）Dubbo

Dubbo 是国内 SOA 框架集大成之作，基本具备一个 SOA 框架应有的所有功能，包括高性能通信、多协议集成、服务注册与发现、服务路由、负载均衡、服务治理等核心功能。作为一款 RPC 架构和 SOA 架构产品，Dubbo 无疑是非常优秀的。但在功能完备性上，API 网关、服务熔断器等核心组件在 Dubbo 中并没有完整体现。

在社区活跃度上，Dubbo 自 2012 年底开始已基本不再更新，但这不影响其在各大互联网公司的应用和扩展，而且从 2017 年开始，阿里巴巴重新启动了 Dubbo 框架的维护工作。

Dubbo 的文档可以说在国内开源框架中算是一流的，非常全面且讲解得比较深入，由于老版本已经非常稳定，所以也不太会出现不一致的情况，学习成本较低。

（2）Spring Cloud

Spring Cloud 是 Spring 家族中新的一员，重点打造面向服务化的功能组件，在功能上服务注册中心、API 网关、服务熔断器、分布式配置中心等组件都能在 Spring Cloud 中找到对应的实现。

在版本更新上，显然 Spring Cloud 也表现得非常活跃。目前 Spring Cloud 在 Github 的托管代码几乎每天都有更新，其发展仍处于高速迭代的阶段。

Spring Cloud 在一定程度上是一种集成型的框架，其内部大量依赖于各种第三方工具和框架，所以文档在体量上自然要比 Dubbo 多很多，文档内容上还算简洁清楚，但是更多的是偏向整合，更深入的使用方法还是需要查看第三方组件的详细文档。

（3）对比与结论

通过对比分析，本书将选择 Spring Cloud 作为实现微服务架构的主体框架。功能的完备性是我们选择 Spring Cloud 的主要原因，表 1-1 中列出了 Dubbo 与 Spring Cloud 功能对比一览，可以看到 Spring Cloud 提供了很多 Dubbo 所不具备但对微服务实现又必不可少的核心组件。

表 1-1　　Dubbo 与 Spring Cloud 功能一览

	Dubbo	Spring Cloud
服务调用方式	RPC	RESTful API
服务注册中心	Zookeeper	Spring Cloud Netflix Eureka
API 网关	无	Spring Cloud Netflix Zuul
服务熔断器	不完善	Spring Cloud Netflix Hystrix
分布式配置	无	Spring Cloud Config
服务跟踪	无	Spring Cloud Sleuth
消息总线	无	Spring Cloud Bus
事件驱动	无	Spring Cloud Stream
安全性	无	Spring Cloud Security

另一个选择Spring Cloud的原因在于服务之间的交互方式。我们知道微服务架构中推崇基于HTTP协议的RESTful风格实现服务间通信，而Dubbo的服务调用基于长连接的RPC实现。采用RPC方式会导致服务提供方与调用方接口产生较强依赖，而且服务对技术敏感，无法做到非常通用。Spring Cloud采用的就是RESTful风格，这方面更加符合微服务架构的设计理念。

Spring Cloud还具备一个天生的优势，因为它是Spring家族中的一员，而Spring在开发领域的强大地位给Spring Cloud起到很好的推动作用。同时，Spring Cloud基于Spring Boot，而Spring Boot目前已经在越来越多的公司得到应用和推广，用来简化Spring应用的框架搭建及开发过程。Spring Cloud也继承了Spring Boot简单配置、快速开发、轻松部署的特点，让原本复杂的架构工作变得相对容易上手。

通过以上分析，相信读者对Dubbo和Spring Cloud有了一个初步的了解。从目前Spring Cloud的被关注度和活跃度上来看，将来很有可能会成为微服务架构的标准框架，本书后续章节将对Spring Cloud的部分核心组件做详细讲解。

1.6 案例分析

本节将通过构建一个精简但又完整的系统来展示微服务架构相关的设计理念和常见实现技术。本案例系统称为订单系统（Order System），试图对互联网应用中最常见的订单业务做抽象。现实环境中订单业务可以非常复杂，该案例的目的在于演示从业务领域分析到系统架构设计再到系统实现的整个过程，不在于介绍具体业务逻辑，所以业务领域建模上做了高度抽象，并不代表实际的应用场景。

按照实施微服务架构的基本思路，服务建模是案例分析的第一步。服务建模包括子域与界限上下文的划分以及服务拆分和集成策略的确定。

OrderSystem包含的业务场景比较简单，用户浏览商品，然后在商品列表中选择想要购买的商品并提交订单；在提交订单的过程中，我们需要对商品和用户账户信息进行验证。从领域的角度进行分析，可以把该系统分成以下3个子域。

- 商品（Product）子域

商品管理，用户可以查询商品以便获取商品详细信息，同时基于商品提交订单；系统管理员可以添加、删除、修改商品信息。

- 订单（Order）子域

订单管理，用户可以提交订单并查询自己所提交订单的当前状态。

- 用户账户（Account）子域

用户管理，我们可以通过注册成为系统用户，同时也可以修改或删除用户信息，并提供账

户有效性验证的入口。

从子域的分类上讲，用户账户子域比较明确，显然应该作为一种通用子域。而订单是 OrderSystem 的核心业务，所以应该是核心子域。至于商品子域，在这里可将其归为支撑子域。从子域之间的上下游关系上看，订单子域需要同时依赖于商品子域和用户子域，商品子域和用户子域之间不存在交互关系。3 个子域的关系如图 1-1 所示。

为了简单起见，我们对每一个子域都提取一个微服务作为示例。基于以上分析，可以把 OrderSystem 简单划分成 3 个微服务：ProductService、OrderService 和 AccountService，图 1-2 展示了 OrderSystem 的基本架构。在图 1-2 中，3 个微服务之间需要基于 REST 进行跨服务之间的交互。

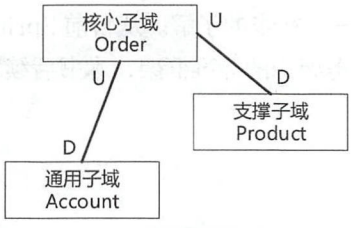


图 1-1　OrderSystem 的 3 个子域

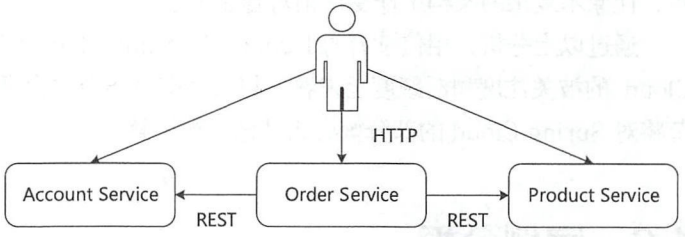


图 1-2　OrderSystem 服务模型

以上 3 个微服务构成了 OrderSystem 的业务主体，而围绕构建一个完整的微服务系统，还需要引入其他很多服务，这些服务从不同的角度为实现微服务架构提供支持。本书中所介绍的关于 Spring Boot、Spring Cloud 和 Docker 的各项核心技术都会在该案例中得到体现。关于该案例的完整代码可参考本书提供的案例资源。

1.7　全书架构

图 1-3 归纳了本书的详细框架。从图中可以看到，全书的行文思路与前面介绍的微服务架构技术体系保持一致。

本章作为全书的第 1 章，主要关注微服务设计原理与架构，通过介绍微服务架构的基础组件和关键要素，引出实现微服务的各项需求并完成技术选型。

在本章的基础上，第 2 章将介绍使用 Spring Boot 构建单个微服务的具体方法和实践。

全书的主要篇幅将围绕 Spring Cloud 展开，从第 3 章到第 8 章将分别介绍微服务架构的各项实现技术，具体包括 Spring Cloud Netflix Eureka 与服务治理、Spring Cloud Netflix Ribbon 与负载均衡、Spring Cloud Netflix Hystrix 与服务容错、Spring Cloud Netflix Zuul 与 API 网关、Spring Cloud Config 与配置中心、Spring Cloud Stream 与事件驱动等核心主题。

图 1-3　全书架构

而从第 9 章到第 12 章则侧重于介绍微服务架构的管理体系，具体包括 Spring Cloud Security 与服务安全、Spring Cloud Sleuth 与服务监控、Spring Test 与服务测试及 Docker 与服务部署等核心主题。

1.8　本章小结

本章作为全书的开篇，围绕微服务架构的定义和特征展开讨论。我们从微服务的基本概念出发，阐述微服务架构的方方面面。微服务架构是一种新型的架构设计模式，是对传统单块系统的改进。我们经历了从单块系统到分布式系统的演变，微服务架构基于分布式系统，但又与

分布式系统存在区别。

微服务架构一方面具备技术、业务和组织上的优势，另一方面也在技术架构和研发过程中存在巨大挑战。微服务架构的实施需要具备一定的前提，而构建微服务架构是一项系统工程，涉及服务建模、服务拆分和服务集成。

本书关注于微服务架构的实现，为了实现微服务架构，我们首先需要明确其在实现上的需求，包括各项基础组件和关键要素。然后，我们就可以根据这些基础组件和关键要素对具体工具进行选型。在本书中，我们把 Spring Cloud 作为构建微服务架构的主体框架。

本书后续章节将围绕微服务架构的技术体系和工程实践展开讨论，在本书中，微服务架构的技术体系分为实现技术和管理体系两个维度。在详细介绍这两个维度之前，下一章内容将首先给出如何使用 Spring Boot 构建单个微服务的方法和实践，这是实现微服务架构的基础。

仅供非商业用途或交流学习使用

第 2 章

使用 Spring Boot 构建服务

本章将使用 Spring Boot 作为实现单个微服务架构的基础框架。目前，Spring Boot 被越来越多的开发团队用于替代原有的 Spring 框架。而在微服务架构中，Spring Boot 也是构成 Spring Cloud 的基础。

Spring Boot 的设计目的是用来简化 Spring 应用程序的初始搭建和开发过程。为了实现这种简化效果，Spring Boot 集成了众多第三方库，并大量使用约定优于配置（Convention Over Configuration）的设计理念，通过特定的方式使得开发人员不再需要定义繁杂而多余的配置内容。

Spring Boot 自诞生以来迅速成为主流的开发框架，其自身也在不断发展。本章内容上会基于 Spring Boot 构建第一个 RESTful 风格的独立服务，并介绍它在数据访问以及消息中间件集成上的具体实现方式。同时，还会利用 Spring Boot 自带的 Actuator 组件来实现内置以及可扩展的系统监控功能。

使用 Spring Boot 开发单个服务的过程非常简单，但简单的背后需要我们深入理解其实现原理。通过对基本原理的分析，我们也将探寻 Spring Boot 所具备的“自动配置”功能的实现机制。

2.1 引入 Spring Boot

微服务架构之间采用轻量级通信的方式进行交互，常见的实现方式是基于 RESTful 风格的 HTTP 请求和响应。图 2-1 展示了典型 RESTful 风格的 HTTP 请求响应的开发流程，可以看到我们需要完成 URL 地址的映射、HTTP 请求的构建、数据的序列化和反序列化（常见的方式就是 JSON），以及实现各个服务自身内部的业务逻辑。

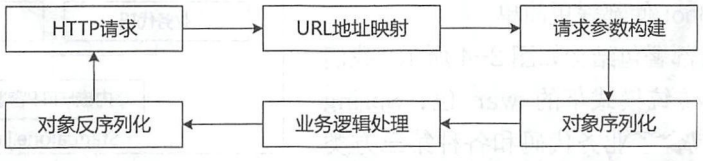


图 2-1　RESTful 风格的 HTTP 请求响应过程

仅供非商业用途或交流学习使用

在引入 Spring Boot 之前，先来回顾一下使用 Spring 完成图 2-1 中各个环节的过程，具体包括使用 web.xml 定义 Spring 的 DispatcherServlet、完成启动 Spring MVC 的配置文件、编写响应 HTTP 请求的 Controller，以及将服务部署到 Tomcat Web 服务器等步骤（见图 2-2）。Spring 框架诞生以来，以上开发流程被广泛采用并形成了一套固定的开发模式。但开发过程同样在不断发展，基于传统的 Spring 框架进行开发逐渐暴露出一些问题，比较典型的就是过于复杂和繁重的配置工作。

若要优化这一套开发过程，有几个值得我们去挖掘的问题，如使用约定优于配置思想的自动化配置、启动依赖项自动管理、简化部署并提供应用监控等。这些优化点推动了以 Spring Boot 为代表的新一代开发框架的诞生。基于 Spring Boot 的开发流程如图 2-3 所示，可以看到它与基于 Spring 的开发流程在配置信息的管理、服务部署和监控等方面有明显不同。

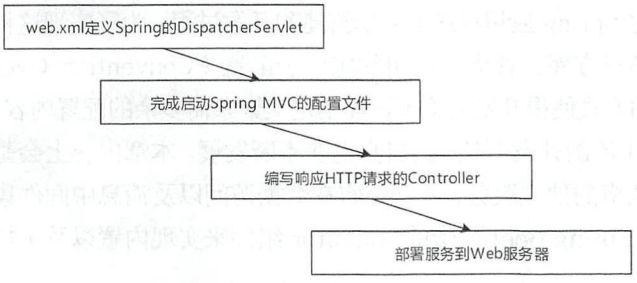


图 2-2　基于 Spring 的开发流程

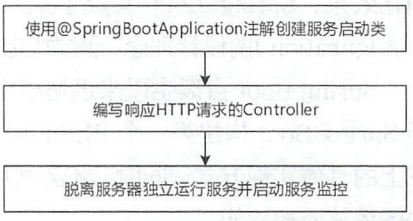


图 2-3　基于 Spring Boot 的开发流程

作为 Spring 家族的新成员，Spring Boot 提供了很多新的特性。例如，支持快速构建项目、不依赖外部容器独立运行、开发部署效率高以及与云平台天然集成。这些特性促使我们选择 Spring Boot 来构建微服务。Spring Boot 的核心优势体现在编码、配置、部署、监控等多个方面。

- Spring Boot 使编码更简单

我们只需要在 Maven 中添加一项依赖并实现一个方法就可以提供微服务架构中所推崇的 RESTful 风格接口。

- Spring Boot 使配置更简单

把 Spring 中基于 XML 的功能配置方式转换为 Java Config，把基于*.properties/*.xml 文件的部署环境配置转换成语义更为强大的*.yml。同时，对常见的各种功能组件均提供了各种默认的 starter 依赖以简化 Maven 配置。

- Spring Boot 使部署更简单

Spring Boot 部署包结构如图 2-4 所示，我们可以看到相较于传统模式下的 war 包，Spring Boot 部署包既包含了业务代码和各种第三方类库，同时也内嵌了 HTTP 容器。这种包结构支持

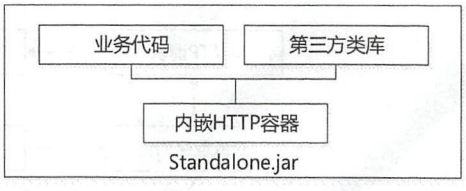


图 2-4　Spring Boot 部署包结构

仅供非商业用途或交流学习使用

java-jar standalone.jar 方式的一键启动，不需要预部署应用服务器，通过默认内嵌 Tomcat 降低对运行环境的基本要求。

- Spring Boot 使监控更简单

基于 spring-boot-actuator 组件，可以通过 RESTful 接口获取 JVM 性能指标、线程工作状态等运行时的信息。例如，可以通过 HTTP GET/env/{name}接口获取系统环境变量，通过 HTTP GET/mapping 接口获取所有 RESTful 服务，通过 HTTP GET/dump 获取线程工作状态，以及通过 HTTP GET/metrics/{name}获取 JVM 性能指标等。

2.2 基于 Spring Boot 的第一个服务

本节将使用 Spring Boot 来构建第一个微服务，完成从环境搭建到最终暴露 RESTful 风格 HTTP 端点（Endpoint）的整个流程。

2.2.1 环境准备

Spring Boot 提供了一系列 starter 工程来简化各种组件之间的依赖关系。例如，在 Spring Boot 中开发基于 RESTful 风格的 HTTP 端点时，我们通常会引入 spring-boot-starter-web 这个工程，打开这个工程会发现里面实际上只定义了如下所示的一些 pom 依赖，可以看到 spring-boot-starter-web 中包括了所有我们能够预见到的组件。如用于构建 Web 请求响应流程的 spring-web 和 spring-webmvc 组件，用于 JSON 序列化和反序列化的 jackson-databind 组件及启动内置 Tomcat 服务器的 spring-boot-starter-tomcat 组件。

- org.springframework.boot:spring-boot-starter
- org.springframework.boot:spring-boot-starter-tomcat
- org.springframework.boot:spring-boot-starter-validation
- com.fasterxml.jackson.core:jackson-databind
- org.springframework:spring-web
- org.springframework:spring-webmvc

引入 spring-boot-starter-web 工程就像引入一个普通的 maven 依赖一样，如下所示。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

本章将引入 Postman 来演示如何通过 HTTP 协议暴露的端点进行远程服务访问。Postman

仅供非商业用途或交流学习使用

提供了强大的 Web API 和 HTTP 请求调试功能，界面简洁明晰，操作方便快捷和人性化。Postman 能够发送任何类型的 HTTP 请求（如 GET、HEAD、POST、PUT 等），并能附带任何数量的参数和 HTTP 请求头（Header）。图 2-5 展示了通过 Postman 访问 HTTP 端点的界面效果，可以看到通过 Postman 可以对各种远程访问场景进行模拟。本书后续章节将大量使用 Postman 来模拟各种 HTTP 请求，Postman 相当于是访问微服务系统的客户端。

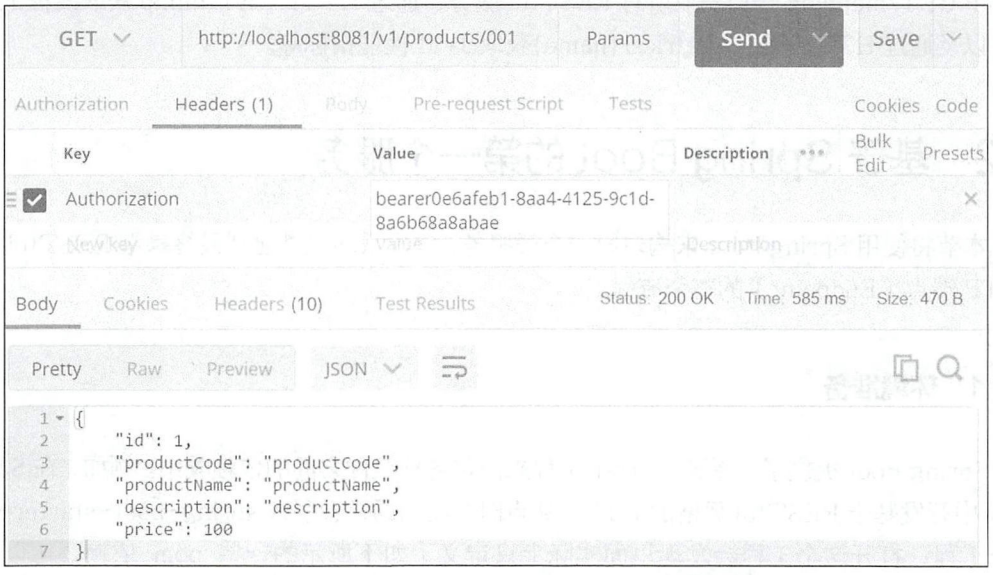


图 2-5　使用 Postman 访问 HTTP 端点效果图

2.2.2　实现 RESTful 服务

在微服务架构中，服务与服务之间使用轻量级的交互方式进行集成，关于服务集成的策略和技术体系可以参考笔者所著《微服务设计原理与架构》中的第 3 章内容。这里所谓的轻量级交互方式指的是基于 RESTful 风格的 HTTP 短连接服务调用方式。在使用 Spring Cloud 构建的微服务架构中，无论是业务服务还是基础类服务，各个服务之间的交互都是采用的 RESTful 风格。因此，本节把实现 RESTful 服务作为演示 Spring Boot 功能的切入点，下面将基于 Spring Boot 构建全书的第一个服务。

1. RESTful 风格简介

REST 提出了一组架构约束条件和原则，满足这些约束条件和原则的应用程序或设计就是 RESTful。现实世界中的事物都可以被认为是一种资源（Resource），我们可以根据这些约束条件和原则设计以资源为中心的服务。REST 中最重要的一条原则就是客户端和服务器之间交

仅供非商业用途或交流学习使用

互的无状态性。从客户端到服务器的每个请求都必须包含理解该请求所必需的信息，无状态请求可以由任何可用服务器实现响应，十分适合微服务架构的运行环境。

在服务器端，应用程序状态和功能可以分为各种资源。每个资源都使用 URI（Universal Resource Identifier，统一资源标识符）得到一个唯一的地址。而在传输协议上使用的就是标准的 HTTP 方法，比如最常见的 GET、PUT、POST 和 DELETE。此外，客户端与服务器端的数据交互涉及序列化问题。所谓序列化（Serialization）就是将对象转化为字节数组，用于网络传输、数据持久化或其他用途；而反序列化（Deserialization）则是把从网络、磁盘等读取的字节数组还原成原始对象。序列化的方式有很多，常见的有文本和二进制两大类。XML 和 JSON 是文本类序列化方式的代表，而二进制实现的方案包括 Google 的 Protocol Buffer 和 Facebook 的 Thrift 等。目前 JSON 是以 Spring Cloud 为代表的微服务架构实现技术中所广泛采用的序列化方式，本书后续的所有示例中都将以 JSON 作为默认的序列化方式。

在 REST 中还有一个重要概念，即 HATEOAS（Hypermedia as the Engine of Application State，基于超媒体的应用状态引擎）。要解释 HATEOAS 这个概念先要解释什么是超媒体。我们已经知道什么是多媒体（Multimedia），以及什么是超文本（Hypertext）。超文本特有的优势是拥有超链接（Hyperlink）。如果把超链接引入多媒体中，就得到了超媒体，因此关键要素还是超链接。使用超媒体作为应用引擎状态，意思就是应用引擎的状态变更由客户端访问不同的超媒体资源驱动。使用 HATEOAS 表现服务请求响应的风格如下所示，可以看到这里多了 _links 属性，其中有一个 self.href 连接指向了当前 user 资源。

```
GET  http://api.example.com/users/tianyalan
Content-Type: application/json
{
  _links: {
    self: {
        href: "/users/tianyalan"
    }
  }
  "id": "tianyalan ",
  "name": "tianyalan",
  "email": "tianyalan @email.com"
}
```

HATEOAS 在 Spring Boot 和 Spring Cloud 中应用也非常广泛，例如，Spring Boot 提供了系统监控组件 Actuator，通过 Actuator 可以获取 Spring Boot 应用程序当前的运行状态。Actuator 组件对外暴露的也是一系列 HTTP 端点，访问这些端点返回的数据跟平时常见的 RESTful 风格有所不同，而这就是 HATEOAS 风格。在微服务架构设计和实现过程中经常会看到这种风格，HATEOAS 不是本书讲解的重点，有兴趣的读者可参考相关资料进一步了解。

仅供非商业用途或交流学习使用

2. Bootstrap 类

使用 Spring Boot 的第一步是创建一个 Bootstrap（启动）类。Bootstrap 类结构简单且比较固化，如下所示。

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HelloApplication {

    public static void main(String[] args) {
        SpringApplication.run(HelloApplication.class, args);
    }
}
```

显然，以上代码中最关键的是一个@SpringBootApplication 注解。Spring Boot 使用@SpringBootApplication 注解来告诉 Spring 容器具备该注解的类是整个 Spring 容器中所有 JavaBean 对象的入口，而具备该注解的类在 Spring Boot 中就是 Bootstrap 类。在上面的代码中 HelloApplication 类就是整个 Spring 容器的 Bootstrap 类。

@SpringBootApplication 注解在指定 Bootstrap 类的同时，还会自动扫描与当前类同级以及子包下的@Component、@Service、@Repository、@Controller、@Entity 等注解并把这些注解对应的类转化为 Bean 对象全部加载到 Spring 容器中管理起来。@SpringBootApplication 注解实际上是由一系列注解组合而成，关于@SpringBootApplication 注解的实现原理将在 2.4 节中具体展开。

我们还注意到在上面的代码中包含一个 main()函数并执行了 SpringApplication.run()方法，该方法的作用就是启动 Spring 容器，并返回 Spring 的 ApplicationContext 对象，我们同样可以根据需要对该 ApplicationContext 对象做相应处理。

3. Controller 类

Bootstrap 类为我们提供了 Spring Boot 应用程序的入口，相当于应用程序已经有了最基本的骨架。接下来就可以添加各种业务相关的访问入口，表现在 Spring Boot 中就是一系列的 Controller 类。这里的 Controller 与 Spring MVC 中的 Controller 在概念上是一致的，最简单的 Controller 类如下所示。

```
@RestController
public class HelloController {

    @RequestMapping("/")
```

仅供非商业用途或交流学习使用


```
    public String index() {
        return "Hello Spring Boot!";
    }
}
```

以上代码中包含了@RestController 和@RequestMapping 这两个注解。我们知道在 Spring MVC 中包含了@Controller 注解，用来标识当前类是一个 Servlet。而@RestController 注解继承自@Controller 注解，它告诉 Spring Boot 这是一个基于 RESTful 风格的 HTTP 端点，并且会自动使用 JSON 实现 HTTP 请求和响应的序列化/反序列化操作。通过这个特性，在构建 REST 服务时可以使用@RestController 注解来代替@Controller 注解以简化开发。另外一个@RequestMapping 注解也与 Spring MVC 保持一致，这里不再赘述。

以下代码展示了一个典型的 Controller，在该 Controller 中我们通过写死的业务代码完成根据商品编号（productCode）获取商品信息的业务流程。这里用到了两层@RequestMapping 注解，在服务层级定义了服务的版本和根路径，分别为 v1 和 products；而在操作级别又定义了具体路径及参数信息。

```
@RestController
@RequestMapping(value = "v1/products")
public class ProductController {

    @RequestMapping(value = "/{productCode}", method = RequestMethod.GET)
    public Product getProduct(@PathVariable String productCode) {

        Product product = new Product();
        product.setId(1L);
        product.setPrice(100F);
        product.setProductCode("product1");
        product.setProductName("Microservice Book");
        product.setDescription("Description of Microservice");

        return product;
    }
}
```

4. 运行 RESTful 服务

现在服务已经开发完成，通过 java-jar 命令可以直接运行 Spring Boot 应用程序。在启动日志中我们发现了以下输出内容（为了显示效果，部分内容做了调整），可以看到自定义的两个 RequestMapping 都已经启动成功并准备接收响应。

```
RequestMappingHandlerMapping : Mapped "{[/]}" onto public
java.lang.String com.tianyalan.springboot.HelloController.index()
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
RequestMappingHandlerMapping : Mapped "{[/v1/products/{productCode}],
methods=[GET]}" onto public com.tianyalan.springboot.product.Product
com.tianyalan.springboot.product.ProductController.getProduct(java.
lang.String)
```

当日志中出现以下信息时代表 Spring Boot 已经启动成功。

```
tianyalan.springboot.HelloApplication  : Started HelloApplication in
10.976 seconds (JVM running for 11.978)
```

这时候通过 Postman 访问“http://localhost:8080/v1/products/product1”端点，可以得到如图 2-6 所示的 HTTP 响应结果。

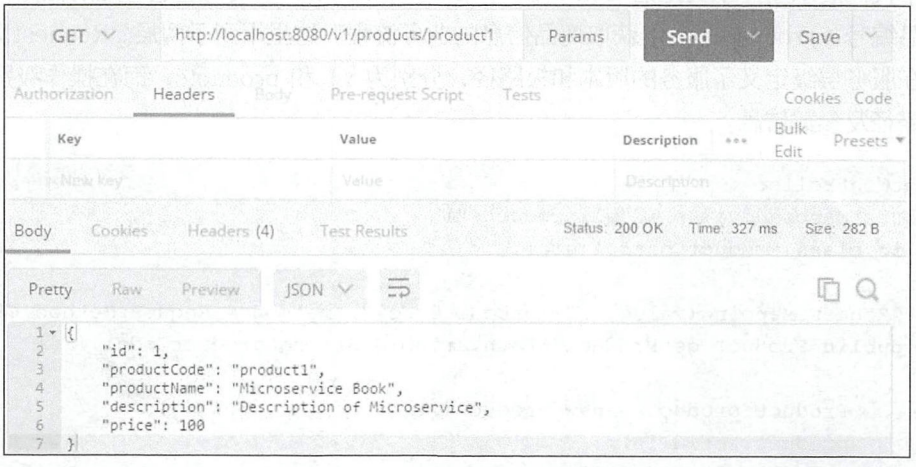


图 2-6　RESTful 服务执行效果图

2.3　Spring Boot 常见功能

开发一个 RESTful 服务，除了实现直接面向客户端或其他服务访问的 Controller，一般还需要实现数据访问。而在分布式系统中，消息通信也是服务之间常见的交互方式。本节将侧重于介绍 Spring Boot 在数据访问和消息中间件集成上的实现方法，并阐述其在系统监控上的特性。

2.3.1　集成 Spring Data

作为服务开发最重要的基础功能之一，数据访问的开发方式在 Spring Boot 中较之 Spring

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

也得到了进一步的简化，并充分发挥了 Spring 家族中另一个重要成员 Spring Data 的作用。Spring Data 是 Spring 家族中专门用于数据访问的抽象框架，其核心理念是支持对所有存储进行资源配置从而实现数据访问。数据访问需要完成领域对象与存储数据之间的映射并对外提供访问入口，Spring Data 基于 Repository 模式并抽象出一套实现该模式的统一数据访问方式。

1. Spring Data 抽象

在日常开发过程中，对数据的查询操作需求远高于新增、删除和修改操作，所以在 Spring Data 中，除了对领域对象提供默认的 CRUD 操作，还重点对查询场景做了高度抽象。我们可以通过@Query 注解直接在代码中嵌入查询语句和条件从而完成领域对象与数据存储的自动映射，提供类似 ORM（Object Relational Mapping，对象关系映射）框架所具有的强大功能。下面就是使用@Query 注解进行查询的典型例子。

```
public interface AccountRepository extends JpaRepository<Account,
    Long> {

    @Query("select a from Account a where a.emailAddress = ?1")
    Customer findByEmailAddress(EmailAddress email);
}
```

方法名衍生查询也是 Spring Data 的查询特色之一，通过在方法命名上直接使用查询字段和参数，Spring Data 就能自动识别相应的查询条件并组装对应的查询语句。如在下面的例子中，通过 findByEmailAndLastname 这样符合普通语义的方法名并在参数列表中按照方法名中参数的顺序和名称（即 email 和 lastname）传入相应的参数，Spring Data 就能自动组装 SQL 语句从而实现衍生查询。

```
public interface AccountRepository extends JpaRepository<Account,
    Long> {

    List<Account> findByEmailAndLastname(EmailAddress email, String
        lastname);
}
```

同时，分页和排序也是查询的常见条件，Spring Data 也提供了 Pageable 和 Sort 抽象。通过构建 Pageable 分页对象和 Sort 查询对象，典型的查询方式如下所示，我们看到可以在代码层面动态实现分页和排序操作。

```
Page<Account> findByLastname(String lastname, Pageable pageable);

List<Account> findByLastname(String lastname, Sort sort);

List<Account> findByLastname(String lastname, Pageable pageable);
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

Spring Data 支持多种 SQL/NoSQL 数据访问，表现在提供了一系列默认的 Repository，包括针对关系型数据库的 JPA/JDBC Repository，针对 MongoDB、Neo4j、Redis 等 NoSQL 对应的 Repository，支持 Hadoop 的大数据访问的 Repository，甚至包括 Spring Batch 和 Spring Integration 在内的系统集成用的 Repository。这里以本书后续章节中介绍的 Spring Data JPA 和 Spring Data Redis 为例讲解 Spring Boot 的具体用法。

2. 集成 Spring Data JPA

在了解了 Spring Data 的各项特性之后，我们将演示如何通过 Spring Data 的这些特性实现数据访问。本节先通过 Spring Data JPA 演示最常见的关系型数据库访问功能，在此之前，需要在 pom 文件中引入 spring-boot-starter-data-jpa 组件，如下所示。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

（1）实体类

首先来看一下 Person 这一常见的业务领域模型，如下代码展示了 Person 实体类。我们看到了代表实体类的@Entity 注解，也看到了用于命名查询的@NamedQuery 注解，该注解用于对某一个查询条件进行命名，如示例中使用“Person.withNameAndAddressNamedQuery”对“select p from Person p where p.name=?1 and address=?2”这一查询进行命名以方便在 Repository 中进行复用。

```
@Entity
@NamedQuery(name = "Person.withNameAndAddressNamedQuery", query =
    "select p from Person p where p.name=?1 and address=?2")
public class Person {

    @Id
    @GeneratedValue
    private Long id;
    private String name;
    private Integer age;
    private String address;

    public Person(Long id, String name, Integer age, String address) {
        super();
        this.id = id;
        this.name = name;
        this.age = age;
        this.address = address;
```

仅供非商业用途或交流学习使用

```
    }

    省略 get/set 操作
}
```

（2）Repository 接口

Repository 接口的定义是 Spring Data JPA 的重点，针对 Person 实体我们可以定义如下的 Repository 接口，其中用到了@Query 注解，也用到了在 Person 实体中定义的“withNameAndAddressNamedQuery”命名查询（即接口定义中的最后一个同名方法）。

```
public interface PersonRepository extends JpaRepository<Person, Long>
{
    List<Person> findByAddress(String address);

    Person findByNameAndAddress(String name,String address);

    @Query("select p from Person p where p.name= :name and
        p.address= :address")
    Person withNameAndAddressQuery(@Param("name")String
        name,@Param("address")String address);

    Person withNameAndAddressNamedQuery(String name,String address);
}
```

（3）Controller 类

正常的分层架构中，一般是Controller层调用Service层，然后Service层再调用Repository层。为了演示的简单，我们直接在 Controller 层中访问 Repository 层的接口，Controller 层代码如下所示，其中的各个端点基本都是对 Repository 层方法的封装，展示了具体的应用场景。

```
@RestController
public class PersonController {

    @Autowired
    PersonRepository personRepository;

    @RequestMapping("/save")
    public Person save(String name,String address,Integer age){
        Person p = personRepository.save(
            new Person(null, name, age, address));
        return p;
    }

    @RequestMapping("/address")
    public List<Person> address(String address){
```

仅供非商业用途或交流学习使用

```
        List<Person> people = personRepository.findByAddress(address)
        return people;
    }

    @RequestMapping("/nameAndAddress")
    public Person nameAndAddress(String name,String address){
        Person people = personRepository
            .findByNameAndAddress(name, address);
        return people;
    }

    @RequestMapping("/nameAndAddressQuery")
    public Person nameAndAddressQuery(String name,String address){
        Person p = personRepository
            .withNameAndAddressQuery(name, address);
        return p;
    }

    @RequestMapping("/nameAndAddressNamedQuery")
    public Person nameAndAddressNamedQuery(String name,String
        address){
        Person p = personRepository
            .withNameAndAddressNamedQuery(name, address);
        return p;
    }

    @RequestMapping("/sort")
    public List<Person> sort(){
        List<Person> people = personRepository
            .findAll(new Sort(Direction.ASC,"age"));
        return people;
    }

    @RequestMapping("/page")
    public Page<Person> page(){
        Page<Person> pagePeople = personRepository
            .findAll(new PageRequest(1, 2));
        return pagePeople;
    }
}
```

（4）配置文件

最后，为了访问数据库，需要在 application.yml 配置文件中添加数据源配置信息，如下所示。

```
spring.datasource.driverClassName=com.mysql.jdbc.Driver
```

仅供非商业用途或交流学习使用

```
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/spring_boot
spring.datasource.username=root
spring.datasource.password=123456
```

3. 集成 Spring Data Redis

如同使用 Spring Data 进行关系型数据库访问一样，使用 Spring Data Redis 的第一步就是连接到 Redis 服务器。想要实现连接，就需要获取 RedisConnection，而获取 RedisConnection 的手段是 RedisConnectionFactory 接口。Spring Data Redis 对 Redis 操作做了封装，提供了一个工具类 RedisTemplate，通过注入 RedisConnectionFactory 到 RedisTemplate 中，该 RedisTemplate 就能获取 RedisConnection。

RedisTemplate 为 Redis 交互提供了一个高级别的抽象。当 RedisConnection 提供低级别的方法来发送和返回二进制值时，RedisTemplate 能够实现序列化和连接过程的自动化管理，从而将用户从这些细节中解放出来。RedisTemplate 提供了丰富的接口来操作 Redis 的特定数据类型，这些接口包括 ValueOperations、ListOperations、SetOperations、ZSetOperations 和 HashOperations 等，分别对应 Redis 中 String、List、Set、ZSet 和 Hash 5 种常见的数据结构。本节将演示通过 ValueOperations 进行 String 类型数据操作的方法。在此之前，我们需要在 pom 文件中引入 spring-boot-starter-data-redis 组件。

（1）实体类

这里用到的实体类与上例中的 Person 在数据字段上保持一致，此处不再赘述。

（2）Repository 类

Spring Data Redis 中的 Repository 类需要注入 RedisTemplate，并提供 ValueOperations 进行具体数据的操作，示例代码如下所示。

```
@Repository
public class PersonRepository {

    @Autowired
    RedisTemplate<Object, Object> redisTemplate;

    @Resource(name = "redisTemplate")
    ValueOperations<Object, Object> valOps;

    public void save(Person person) {
        valOps.set(person.getId(), person);
    }

    public Person getPerson() {
        return (Person) valOps.get("1");
    }
```

仅供非商业用途或交流学习使用

```
}
```

这里的 ValueOperations 依赖于 RedisTemplate，而在 RedisTemplate 中可以对 RedisConnectionFactory 和序列化方式进行设置。在 Spring Boot 中，用于构建 RedisTemplate 的代码一般会放到 Bootstrap 类中。

（3）Bootstrap 类

Bootstrap 类如下所示，可以看到这里有一个 redisTemplate()方法用于构建 RedisTemplate 并初始化用户数据存储的序列化方式。

```
@SpringBootApplication
public class RedisApplication {

    public static void main(String[] args) {
        SpringApplication.run(RedisApplication.class, args);
    }

    @Bean
    @SuppressWarnings({ "rawtypes", "unchecked" })
    public RedisTemplate<Object, Object>
        redisTemplate(RedisConnectionFactory
            redisConnectionFactory)
        throws UnknownHostException {

        RedisTemplate<Object, Object> template = new
            RedisTemplate<Object, Object>();
        template.setConnectionFactory(redisConnectionFactory);

        Jackson2JsonRedisSerializer jackson2JsonRedisSerializer =
            new Jackson2JsonRedisSerializer(Object.class);
        ObjectMapper om = new ObjectMapper();
        om.setVisibility(PropertyAccessor.ALL,
            JsonAutoDetect.Visibility.ANY);

        om.enableDefaultTyping(
            ObjectMapper.DefaultTyping.NON_FINAL);
        jackson2JsonRedisSerializer.setObjectMapper(om);

        template.setValueSerializer(jackson2JsonRedisSerializer);
        template.setKeySerializer(new StringRedisSerializer());

        return template;
    }
}
```

仅供非商业用途或交流学习使用

（4）Controller 类

最后是 Controller 类，代码如下所示，就是对 PersonRepository 的简单封装。

```
@RestController
public class PersonController {

    @Autowired
    PersonRepository personRepository;

    @RequestMapping("/setPerson")
    public void setPerson(){
        Person person = new Person("1","tianyalan", 32);
        personRepository.save(person);
    }

    @RequestMapping("/getPerson")
    public Person getPerson(){
        return personRepository.getPerson();
    }
}
```

（5）application.yml

配置文件 application.yml 可以全部采用默认配置，即 Redis 服务器会以默认的 6379 端口号进行启动。

4. 集成 Spring Data Mongodb

最后来演示如何通过 Spring Data Mongodb 组件访问 Mongodb。Mongodb 是一种文档型的 Nosql 数据库，它提供了很多有用的查询方法。在演示 Spring Data Mongodb 的具体功能之前，需要先在 pom 文件中引入 spring-boot-starter-data-mongodb 组件。

（1）实体类

本示例的 Person 类定义如下。这里用了@Document 注解，表明该类对应 Mongodb 中的一个文档，同时使用@Field 注解来显式声明文档中的字段，注意到这里的 locations 字段是一个 Location 对象的集合。

```
@Document
public class Person {

    @Id
    private String id;
    private String name;
    private Integer age;
    @Field("locs")
```

```
    private Collection<Location> locations = new
        LinkedHashSet<Location>();

    public Person(String name, Integer age) {
        super();
        this.name = name;
        this.age = age;
    }

    省略 get/set 操作
}


public class Location {

    private String place;
    private String year;

    public Location(String place, String year) {
        super();
        this.place = place;
        this.year = year;
    }

    省略 get/set 操作
}
```

（2）Repository 类

PersonRepository 接口扩展了 Spring Data 中的 MongoRepository 接口，代码如下所示。可以看到该接口的风格与使用 Spirng Data JPA 定义的 Repository 接口非常一致，同样也提供了@Query 注解来定义查询条件。

```
public interface PersonRepository extends MongoRepository<Person,
    String> {

    Person findByName(String name);

    @Query("{'age': ?0}")
    List<Person> withQueryFindByAge(Integer age);
}
```

（3）Controller 类

Controller 类如下所示，除 save()方法比较复杂之外，其他的方法都是对 Repository 的简单封装。

```
@RestController
public class PersonController {

    @Autowired
    PersonRepository personRepository;

    @RequestMapping("/save")
    public Person save(){
        Person  p = new Person("tianyalan",32);
        Collection<Location> locations =  new
            LinkedHashSet<Location>();

        Location hangzhou = new Location("杭州","2002");
        locations.add(hangzhou);
        p.setLocations(locations);

        return personRepository.save(p);
    }

    @RequestMapping("/queryByName")
    public Person queryByName(String name){
        return personRepository.findByName(name);
    }

    @RequestMapping("/queryByAge")
    public List<Person> queryByAge(Integer age){
        return personRepository.withQueryFindByAge(age);
    }
}
```

（4）application.yml

配置文件 application.yml 同样全部采用了默认配置，即 Mongodb 服务器以默认的端口号 27017 进行启动。

2.3.2 集成消息中间件

Spring Boot 同样能以简单的方式完成与各种消息中间件的集成。关于与消息中间件的集成，Spring Cloud 中提供了专门的事件驱动组件 Spring Cloud Stream，我们将在第 8 章中对 Spring Cloud Stream 进行详细介绍。这里无意对各种消息中间件做过多展开，仅以 RabbitMQ 为例进行简单介绍。

RabbitMQ 是 AMQP（Advanced Message Queuing Protocol，高级消息队列协议）规范

的一种实现，用于在分布式系统中实现对消息的存储转发（Store and Forward）。Spring Boot 集成 RabbitMQ 非常简单，提供了专门的 spring-boot-starter-amqp 组件支持各种消息处理。

（1）队列配置

在使用 RabbitMQ 发送和消费消息之前，需要指定队列信息，可以通过专门的配置类来统一管理。例如，想要声明一个“tianyalan-queue”的队列，可以使用如下代码。

```
@Configuration
public class RabbitConfig {

    @Bean
    public Queue tianyalanQueue(){
        return new Queue("tianyalan-queue");
    }
}
```

（2）BootStrap 类与消息生产者

在 Spring 框架中提供了一个 RabbitTemplate 工具类对通过 AMQP 与 RabbitMQ 进行交互的过程进行了封装。与其他 Template 工具类一样，RabbitTemplate 的背后同样是通过 ConnectionFactory 获取与 RabbitMQ 的连接，但在 Spring Boot 中，这部分工作通过自动配置机制已经得以实现，一般情况下我们只需要注入 RabbitTemplate 即可。BootStrap 类的代码如下所示。

```
@SpringBootApplication
public class AmqpApplication implements CommandLineRunner {

    @Autowired
    RabbitTemplate rabbitTemplate;

    public static void main(String[] args) {
        SpringApplication.run(AmqpApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        rabbitTemplate.convertAndSend("tianyalan-queue", "message
            from rabbitmq");
    }
}
```

我们应该注意到了 AmqpApplication 的特殊性，因为它实现了 CommandLineRunner 接口。实际应用中，我们有时候会需要在 Spring Boot 应用程序启动的时候就去执行一些操作，为了实现这样的需求，Spring Boot 提供了一个方案，即 CommandLineRunner 接口。当 Spring 的

ApplicationContext 初始化完成之后，应用程序中存在的所有 CommandLineRunner 都会被执行。CommandLineRunner 的接口定义如下所示。

```
public interface CommandLineRunner {
    void run(String... args) throws Exception;
}
```

Spring Boot 应用程序在启动后，会遍历 CommandLineRunner 接口的实例并运行它们的 run()方法。我们也可以利用@Order 注解来规定所有 CommandLineRunner 实例的运行顺序。在这个例子的 run()方法中，我们通过 RabbitTemplate 提供的 convertAndSend()方法完成消息的发送，即当该 Spring Boot 应用程序启动成功之后，系统会自动往“tianyalan-queue”队列中发送一条“message from rabbitmq”消息。

（3）消息消费者

接下来是消息消费者，通过@RabbitListener 注解可以把一个普通的服务转变成消息消费的入口。消息消费者代码如下所示，运行效果相当于当“tianyalan-queue”队列中新来一条消息时，将自动触发 receiveMessage()这一回调函数。

```
@Component
public class Receiver {

    @RabbitListener(queues = "tianyalan-queue")
    public void receiveMessage(String message) {
        System.out.println("Received:" + message);
    }
}
```

（4）application.yml

可以使用 Spring Boot 默认的配置完成消息生产和消费，如果需要改变默认配置，也可以在 application.yml 添加类似如下的配置项。

```
spring.rabbitmq.host=127.0.0.1
spring.rabbitmq.port=5672
spring.rabbitmq.username=tianyalan
spring.rabbitmq.password=123456
```

2.3.3 系统监控

本节将介绍 Spring Boot 自带的系统监控功能，系统监控在 Spring 中是缺失的，而 Spring Boot 考虑到了这方面的需求并提供了 Actuator 组件。Actuator 是 Spring Boot 提供的一种集成功能，可以实现对应用系统的运行时状态管理、配置查看以及相关功能统计。

1. 初始化 Spring Boot 监控

初始化 Spring Boot 监控需要引入 Spring Boot Actuator 组件，而 Spring Boot Actuator 组件又依赖于 Spring HATEOAS 组件，所以需要在 pom 中添加如下两个 maven 依赖。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.hateoas</groupId>
    <artifactId>spring-hateoas</artifactId>
</dependency>
```

当启动 Spring Boot 应用程序时，启动日志里会自动添加 autoconfig、dump、beans、actuator、health 等多个 HTTP 的端点。当访问 http://localhost:8080/actuator 端点时会得到如下结果，这种响应结果就是 HATEOAS 风格的 HTTP 响应。

```
{
    "links":[
        {
            "rel":"self",
            "href":"http://localhost:8080/actuator"
        },
        {
            "rel":"metrics",
            "href":"http://localhost:8080/metrics"
        },
        {
            "rel":"configprops",
            "href":"http://localhost:8080/configprops"
        },
        {
            "rel":"autoconfig",
            "href":"http://localhost:8080/autoconfig"
        },
        {
            "rel":"beans",
            "href":"http://localhost:8080/beans"
        },
        {
            "rel":"trace",
            "href":"http://localhost:8080/trace"
        },
```

```
        {
            "rel":"env",
            "href":"http://localhost:8080/env"
        },
        {
            "rel":"info",
            "href":"http://localhost:8080/info"
        },
        {
            "rel":"health",
            "href":"http://localhost:8080/health"
        },
        {
            "rel":"mappings",
            "href":"http://localhost:8080/mappings"
        },
        {
            "rel":"dump",
            "href":"http://localhost:8080/dump"
        }
    ]
}
```

以上是 Spring Boot Actuator 默认提供的端点列表，各个端点的路径和描述参考表 2-1。

表 2-1　　Spring Boot Actuator 端点列表

http 方法	路　径	描　述
get	/autoconfig	提供了一份自动配置报告，描述当前环境中加载的自动配置项
get	/configprops	描述配置属性（包含默认值）如何注入 Bean
get	/beans	描述应用程序上下文里全部的 Bean 以及它们的关系
get	/dump	获取线程活动的快照
get	/env	获取全部环境属性
get	/env/{name}	根据名称获取特定的环境属性值
get	/health	报告应用程序的健康指标，这些值由 HealthIndicator 的实现类提供
get	/info	获取应用程序的定制信息，这些信息由 info 打头的属性提供
get	/mappings	描述全部的 URI 路径，以及它们和 Controller（包含 Actuator 端点）的映射关系
get	/metrics	报告各种应用程序度量信息，如内存用量和 HTTP 请求计数
get	/metrics/{name}	报告指定名称的应用程序度量值
post	/shutdown	关闭应用程序，要求 endpoints.shutdown.enabled 设置为 true
get	/trace	提供基本的 HTTP 请求跟踪信息（包括时间戳、HTTP 头等）

访问表 2-1 中的各个端点可以获取监控信息，例如访问 http://localhost:8080/health 可以用得到当前服务器的基本状态，如下所示。

```
{
    "status":"UP",
    "diskSpace":{
        "status":"UP",
        "total":73400315904,
        "free":2875985920,
        "threshold":10485760
    }
}
```

读者可以访问各个端点获取自己感兴趣的监控信息。如果 Spring Boot Actuator 默认提供的端点信息不能满足需求，也可以对其进行修改和扩展。常见实现方案有两种：一种是扩展现有的监控端点，另一种是自定义新的监控端点。

2. 扩展 Spring Boot 监控端点

假如需要对表 2-1 中的/health 端点进行扩展以添加当前用户账户(Account)的相关信息，我们可以实现 HealthIndicator 接口。HealthIndicator 接口只有一个 health()方法返回 Health 对象，通过在 Health 对象中添加新的监控配置项就可以实现扩展，示例代码如下所示，我们添加了“accountcount”和“accountstatus”这两个监控项。为了演示方便，这里对这两个监控项的值做了写死处理，我们可以根据需要设置相应的运行时状态信息。

```
@Component
public class AccountIndicator implements HealthIndicator {

    @Override
    public Health health() {
        return new Health.Builder()
            .withDetail("accountcount", 10)
            .withDetail("accountstatus", "on")
            .up().build();
    }
}
```

重新运行 Spring Boot 应用程序，然后访问 http://localhost:8080/health 端点就会得到如下信息，可以看到在返回的 JSON 数据中多了一项“account”并且包含代码中设置的监控项信息。

```
{
    "status": "UP",
    "account": {
```

```
        "status": "UP",
        "accountcount ": 10,
        "accountstatus ": "on"
    },
    "diskSpace":{
        "status":"UP",
        "total":73400315904,
        "free":2875985920,
        "threshold":10485760
    }
}
```

3. 自定义 Spring Boot 监控端点

有时候扩展现有端点并不一定能够满足需求，自定义 Spring Boot 监控端点是更灵活的方法。例如，当需要提供一个监控端点以获取当前主机的计算机名称和用户信息时，就可以实现一个独立的 UserEndPoint，代码如下所示。UserEndPoint 扩展了 AbstractEndpoint 类，并通过系统环境变量获取所需监控信息。

```
@ConfigurationProperties(prefix="endpoints.usersystem")
public class UserEndPoint extends AbstractEndpoint<Map<String,
    Object>> {

    public UserEndPoint (){
        super("usersystem");
    }

    @Override
    public Map<String, Object> invoke() {
        Map<String,Object> result= new HashMap<>();
        Map<String, String> map = System.getenv();
        result.put("username",map.get("USERNAME"));
        result.put("computername",map.get("COMPUTERNAME"));
        result.put("userdomain",map.get("USERDOMAIN"));
        return result;
    }
}
```

当完成自定义的 UserEndPoint 端点时，需要把它放到 Spring Boot 配置系统中，方法如下。

```
@Configuration
public class EndPointAutoConfig {

    @Bean
    public Endpoint<Map<String, Object>> userEndPoint() {
```

```
        return new UserEndPoint();
    }
}
```

重新启动 Spring Boot 应用程序，在启动日志中会发现如下信息，这说明/usersystem 自定义端点已经生效。

```
EndpointHandlerMapping: Mapped "{[/usersystem || /usersystem.json],
methods=[GET],produces=[application/json]}" onto public java.lang.Object org.
springframework.boot.actuate.endpoint.mvc. EndpointMvcAdapter.invoke()
```

现在访问 http://localhost:8088/usersystem 就能获取如下监控信息。

```
{
    "username": "tianyalan",
    "userdomain": "DESKTOP-6EAN1H4",
    "computername": "DESKTOP-6EAN1H4"
}
```

2.4 Spring Boot 基本原理

通过@SpringBootApplication 注解并配合各种内置的 starter 代码工程，我们可以利用 Spring Boot 快速打造一个复杂系统构建过程中所需的各种功能体系。本节将分析@SpringBootApplication 注解背后的实现原理。

我们已经在本章前面各节内容中看到，基于 Spring Boot 的开发模式非常简单。显然，这种简单性得以实现的关键在于有一个强大的注解@SpringBootApplication，@SpringBootApplication 注解的定义如下，实际上它是由 3 个注解组合而成：@Configuration、@EnableAutoConfiguration 和@ComponentScan。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Configuration
@EnableAutoConfiguration
@ComponentScan
public @interface SpringBootApplication
```

在 Spring 中，@Configuration 注解比较常见，提供了 JavaConfig 配置类实现。而@ComponentScan 则扫描@Component 等注解，把相关 Bean 定义批量加载到 IoC 容器中。这里需要重点分析的是@EnableAutoConfiguration 注解，该注解的定义如下，我们看到该注解中通过

@Import 注解引入了 EnableAutoConfigurationImportSelector 类。@Import 注解在 Spring 中用于导入 JavaConfig 配置类，而在这里导入的就是 EnableAutoConfigurationImport Selector 类。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(EnableAutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration
```

从类名上看，EnableAutoConfigurationImportSelector 类是一种选择器，负责从各种配置项中找到需要导入的具体配置类。EnableAutoConfigurationImportSelector 类的结构如图 2-7 所示，其所依赖的最关键组件是 SpringFactoriesLoader。

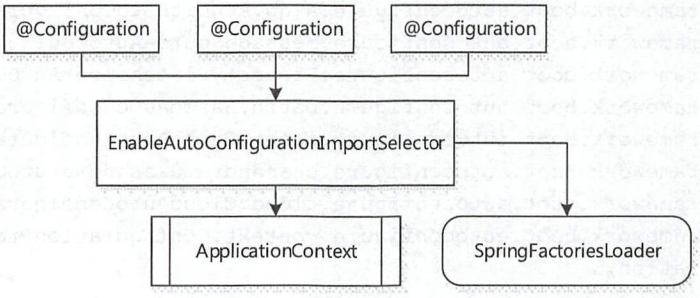


图 2-7　EnableAutoConfigurationImportSelector 结构

要想理解 SpringFactoriesLoader 类，首先需要了解 JDK 中的 SPI（Service Provider Interface，服务提供者接口）机制。JDK 提供了实现服务查找的一个工具类 java.util.Service Loader 来实现 SPI 机制。当服务提供者提供了服务接口的一种实现之后，我们可以在 jar 包的 META-INF/services/ 目录创建一个以服务接口命名的文件，该文件里配置着一组 Key-Value，用于指定服务接口与实现该服务接口具体实现类的映射关系。而当外部程序装配这个 jar 包时，就能通过该 jar 包 META-INF/services/ 目录中的配置文件找到具体的实现类名，并装载实例化，从而完成模块的注入。SPI 机制的结构以及与 API 的关系如图 2-8 所示。

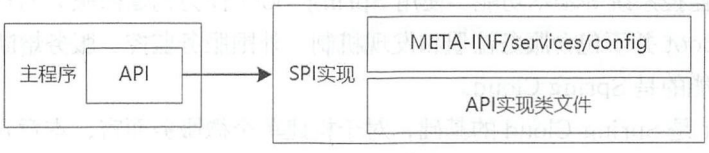


图 2-8　SPI 结构图

SPI 提供了一种约定，基于该约定就能很好的找到服务接口的实现类，而不需要在代码里

硬编码指定。SpringFactoriesLoader 类似这种 SPI 机制，只不过以服务接口命名的文件是放在 META-INF/spring.factories 文件夹下，对应的 Key 为 EnableAutoConfiguration。SpringFactoriesLoader 会查找所有 META-INF/spring.factories 文件夹中配置文件，并把 Key 为 EnableAutoConfiguration 所对应的配置项通过反射实例化为配置类并加载到容器。以下就是 Spring Boot 中所使用的 spring.factories 配置文件片段，可以看到 EnableAutoConfiguration 中包含了各式各样的配置项，这些配置项在 Spring Boot 启动过程中都能够通过 SpringFactoriesLoader 加载到运行时环境，从而实现自动化配置。

```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdmin
JmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.MessageSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.PropertyPlaceholderAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\
org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration,\
org.springframework.boot.autoconfigure.context.ConfigurationProperties
AutoConfiguration,\
...
```

以上就是 Spring Boot 实现自动配置的基本过程和原理。当然，@SpringBootApplication 注解也可以基于外部配置文件加载配置信息。基于约定优于配置思想，Spring Boot 在加载外部配置文件的过程中大量使用了默认配置。

2.5 本章小结

针对一般的服务化开发模式，Spring Boot 为我们提供了 RESTful API、数据访问、消息传递以及内置的监控系统等基本功能。使用 Spring Boot 作为构建微服务的轻量级框架是合适的，但 Spring Boot 并不包含服务注册和发现机制、外围服务监控、服务熔断和服务治理等功能，提供这些功能的是 Spring Cloud。

Spring Boot 是 Spring Cloud 的基础，对于构建单个微服务而言，本章介绍的内容适用于所有的服务构建。从第 3 章开始，将开始全面介绍 Spring Cloud 中的各个核心组件，并给出这些组件的功能特性和使用方法。

第 3 章

Spring Cloud Netflix Eureka 与服务治理

在微服务架构中，服务与服务之间通过交互完成业务链路的构建，这种服务交互采用的是轻量级的 RESTful 风格 API 调用方式。在服务调用之前，首先需要发现（Discovery）服务，即解决在分布式集群环境下如何找到合适的服务实例问题。服务发现和调用构成了微服务交互的基础，微服务架构下的服务发现和调用可以参考图 3-1 中的整体流程，其中，实线部分代表服务调用流程，而虚线部分则包含了服务的注册（Registration）和发现过程。

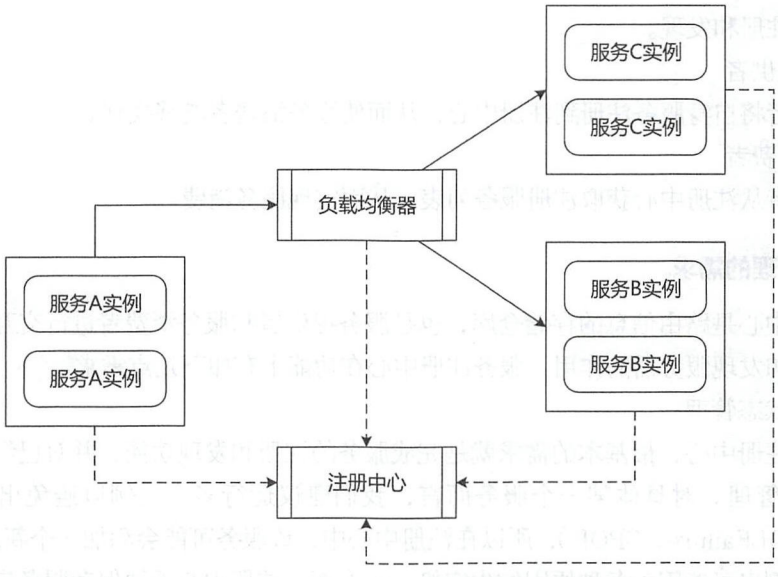


图 3-1　服务发现和调用流程图

在图 3-1 中存在 3 个微服务，即服务 A、服务 B 和服务 C，每个微服务都是多实例部署在集群环境中。一方面，服务 B 和服务 C 对于服务 A 而言是服务的提供者，作为消费者的服务 A 需要通过负载均衡器分别找到服务 B 和服务 C 的实例并完成服务调用。另一方面，这 3 个微服务都需要到注册中心注册，以便负载均衡器能够从注册中心获取各个服务的定义信息。本章内容将围绕基于注册中心的服务治理（Service Governance）机制展开讨论，负载均衡的相关内容将在下一章中介绍。

3.1 服务治理解决方案

在微服务架构中，服务治理可以说是最为关键的一个要素，因为各个微服务需要通过服务治理实现自动化的注册和发现。本章将重点介绍 Spring Cloud 中的服务治理组件 Spring Cloud Netflix Eureka，但在此之前，我们很有必要梳理服务治理的基本需求、设计模型，以及相应的解决方案。

3.1.1 服务治理的需求和模型

当具备服务注册中心之后，服务治理涉及的角色包括如下 3 种。

- 注册中心

提供服务注册和发现。

- 服务提供者

服务提供者将自身服务注册到注册中心，从而使服务消费者能够找到。

- 服务消费者

服务消费者从注册中心获取注册服务列表，从而实现服务消费。

1. 服务治理的需求

服务注册中心是路由信息的存储仓库，也是服务提供者和服务消费者进行交互的媒介，充当着服务注册和发现服务器的作用。服务注册中心在功能上有如下几点需求。

（1）服务状态管理

作为服务注册中心，最基本的需求就是完成服务的注册和发现功能，并对已经存在的各个服务进行统一管理。对具体某一个服务而言，我们建议运行多个实例以避免出现单点失败（Single Point Of Failure，SPOF），所以在注册中心中，该服务可能会添加一个新的运行实例，而运行中的实例也可能因为各种原因发生宕机。一方面，注册中心需要保存服务定义信息；另一方面，当服务运行状态发生变化时，注册中心需要确保这些状态变更都能得到有效维护。

从架构设计上讲，状态变更管理可以采用发布-订阅（Publish Subscribe，Pub-Sub）模式，即服务提供者可以根据服务定义发布服务，服务消费者可以通过对自己感兴趣的服务进行订阅并获取包括服务地址在内的各项元数据。发布-订阅功能还体现在状态变更推送上，即当注册中心服务定义发生变化时，主动推送变更到该服务的消费者。

由于服务提供者和服务消费者同时依赖于注册中心，这就需要确保注册中心内部的数据一致性，即在任何时候服务提供者和消费者都应该看到同一份数据。作为发布-订阅模式的一种

实现，注册中心的这些功能都是与该模式的基本原理保持一致。

（2）高可用

为了确保服务高可用性，一般也需要注册中心本身保持高可用性，也就意味着注册中心本身需要构建对等集群。所谓对等集群（Peer-to-peer Server Cluster），是指集群中所有的服务器都提供同样的服务，所以在对等集群中，客户端只要连接一台服务器完成服务注册和发现即可，任何一台服务器宕机都不影响客户端正常使用。

（3）易伸缩

当把所有的服务定义集中在服务注册中心且单个服务通过集群进行访问时，我们就需要考虑伸缩性问题。一般通过负载均衡机制可以实现服务访问的横向伸缩，而常见的负载均衡策略包括服务器端负载均衡和客户端负载均衡。采用何种具体的负载均衡策略取决于架构设计，但注册中心本身保存了服务列表，通过注册中心可以简单实现服务器端的负载均衡。如果是客户端负载均衡，注册中心也需要能够提供实时有效的服务列表。

（4）方便集成

注册中心是一种服务器组件，而服务的提供者和消费者都是注册中心的客户端。这些客户端和注册中心之间需要进行交互，由于涉及服务的发布和订阅、服务访问中的负载均衡等机制，需要确保交互过程简单而高效。同时，考虑到异构系统之间的交互需求，注册中心作为一种平台化解决方案也应该提供多种客户端技术的集成支持。

2. 注册中心模型

正如上文中的分析，访问注册中心的客户端程序一般会嵌入在服务提供者和服务消费者内部。在服务运行时，服务提供者的注册中心客户端程序会向注册中心注册自身提供的服务，而服务消费者的注册中心客户端程序则从注册中心获取当前订阅的服务信息状态。

同时，为了提高服务路由的效率和容错率，服务消费者可以配备缓存机制以加速服务路由，更重要的是当服务注册中心不可用时，服务消费者可以利用本地缓存路由实现对现有服务的可靠调用。图3-2就展示了这一设计思路。

基于以上分析，我们可以对注册中心进行抽象得到如图3-3所示的模型图，这里的客户端既可以是服务提供者，也可以是服务消费者。下面将讨论如何实现该模型。

3.1.2 服务治理的基本方案

在本节中，我们将首先介绍服务治理的基本工作流程和实现策略，并给出目前市面上主流的服务治理实现工具。

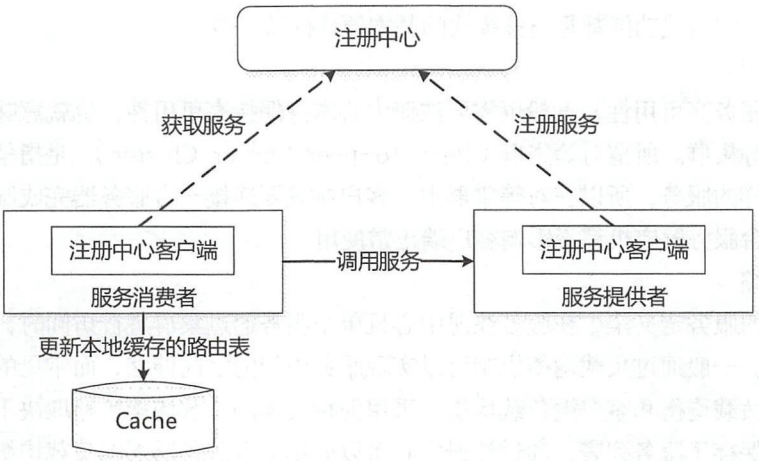


图 3-2　注册中心客户端与注册中心交互过程

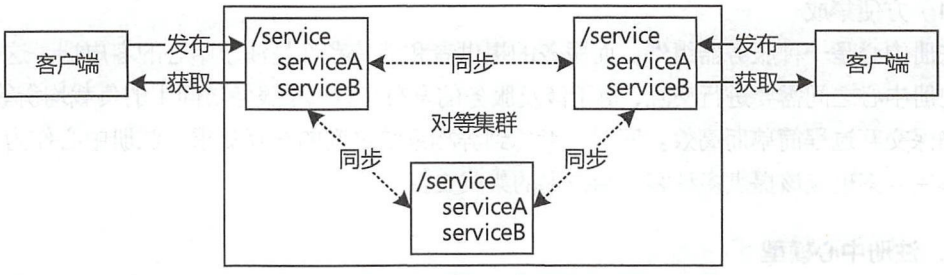


图 3-3　注册中心模型

1. 服务治理流程和策略

基于注册中心各项特性所采取的注册中心基本实现流程如图 3-4 所示，该图中服务的提供者和消费者可以使用一定的通信机制与注册中心服务器建立连接并维持心跳检测，通过注册中心提供的操作接口分别完成发布和动态更新服务定义、获取指定服务地址列表、取消服务发布、获取服务地址更新等功能。

在图 3-4 中，基本的工作流程通过操作语义即可理解，但有一个步骤值得具体展开，那就是服务的消费者如何获取服务列表，实现上通常有如下两种方案。

（1）轮询机制

轮询（Polling）机制是一种主动拉取策略，即服务的消费者定期调用注册中心提供的服务获取接口获取最新的服务列表并更新本地缓存。轮询机制实际上就是一个定时程序，需要考虑定时的频率以确保数据同步的时效性。

（2）监听和通知机制

服务监听机制确保消费者能够实时监控服务更新状态，是一种被动接收变更通知的实现方

案，通常采用监听器（Watcher）及回调（Callback）机制。如图 3-5 所示，假定以一种树状结构来展示注册中心的内部结构，可以看到客户端可以对位于/services 根目录下的服务地址和具体的服务定义节点添加监听器，当这些节点发生变化时（如图 3-5 中 Service2 节点表示取消注册，Service5 和 Service6 节点表示新增注册），注册中心就能触发各个客户端监听器中的回调函数以确保更新通知到每一个客户端。显然，使用监听和通知机制具备实时的数据同步效果。

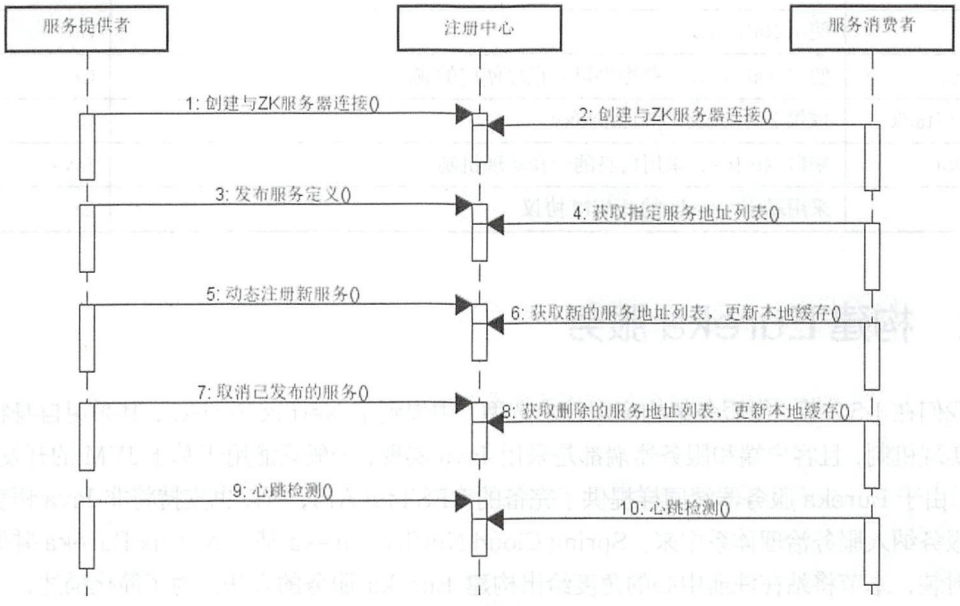


图 3-4　服务发布订阅流程

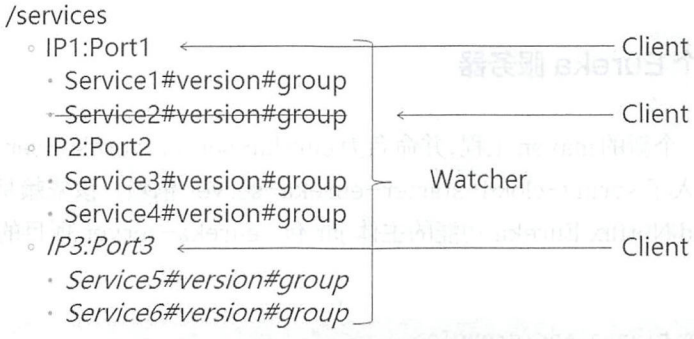


图 3-5　注册中心变更通知机制示意图

2. 服务治理实现工具

以上关于服务治理解决方案的讨论为我们提供了理论基础，根据这些理论基础，目前市面上存在一批主流的实现工具，包括常见的 Zookeeper、Consul 等，各个工具的实现机制和实现

语言如表3-1所示。在本书中关于服务治理的具体实现方案将围绕Spring Cloud 旗下的Spring Cloud Netflix Eureka 展开讨论。

表 3-1　　服务治理实现方案

产　品	实 现 机 制	实 现 语 言
Zookeeper	提供强一致性，使用基于 Paxos 协议的 ZAB 协议	Java
Etcd	使用 Raft 协议	Go
Consul	使用 Raft 协议，并提供可以直接使用的成品	Go
SmartStack	依赖 Zookeeper 和 Haproxy	Go
Eureka	来自 Netflix，采用自身的一套实现机制	Java
Serf	采用基于 Gossip 的 SWIM 协议	Go

3.2 构建 Eureka 服务

我们在 1.5.2 节“微服务架构实现技术选型”中提到了 Netflix Eureka，其采用自身特有的一套实现机制，且客户端和服务器端都是采用 Java 实现，一般只能用于基于 JVM 的开发环境。当然，由于 Eureka 服务器端同样提供了完备的 RESTful API，所以也支持将非 Java 语言构建的微服务纳入服务治理体系中来。Spring Cloud Netflix Eureka 基于 Netflix Eureka 并做了一定的封装，本节将站在注册中心的角度给出构建 Eureka 服务的方法。为了简化描述，下文中将用 Eureka 代表 Spring Cloud Netflix Eureka。

3.2.1 构建单个 Eureka 服务器

本节将创建一个新的 maven 工程，并命名为 eureka-server。eureka-server 是一个 Spring Boot 项目，它引入了 spring-cloud-starter-eureka-server 依赖，该依赖是 Spring Cloud 中实现 Spring Cloud Netflix Eureka 功能的主体 jar 包。eureka-server 项目的 pom 文件的部分内容如下所示。

```
<groupId>com.tianyalan</groupId>
<artifactId>eureka-server</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>

<name>Eureka Server</name>
<dependencies>
```

```
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-eureka-server</artifactId>
    </dependency>
</dependencies>
```

引入 maven 依赖之后，就可以创建运行 Eureka 服务器的 Bootstrap 类。在示例代码中，我们把该 Bootstrap 类命名为 EurekaServerApplication，代码如下所示。

```
package com.tianyalan.eurekaserver;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {
    public static void main(String[] args) {

        SpringApplication.run(EurekaServerApplication.class, args);
    }
}
```

在上面的代码中，从结构上讲就是一个普通的 Bootstrap 类，但有一个注解值得注意，即@EnableEurekaServer。在 Spring Cloud 中，包含@EnableEurekaServer 注解的服务意味着它是一个 Eureka 服务器组件。

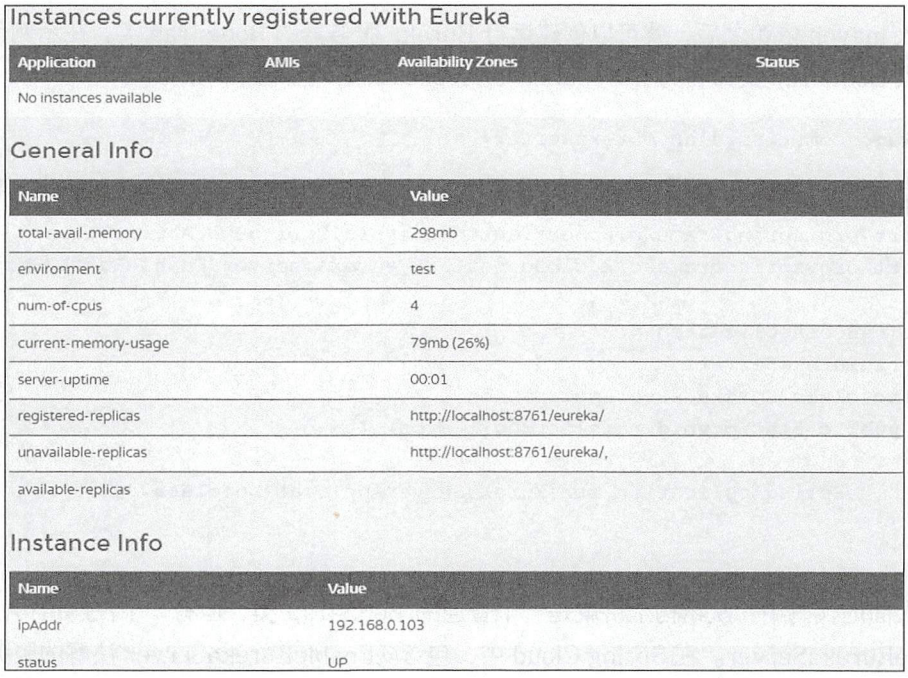
在本地环境运行 EurekaServerApplication 类并访问 http://localhost:8761/，将得到如图 3-6 所示的 Eureka 服务监控页面，这意味着 Eureka 服务器已经启动成功。在图 3-6 中，我们看到了运行该 Eureka 服务的主机信息，以及当前注册到该注册中心的服务实例信息。因为 Eureka 服务是我们构建的第一个微服务，所以目前还没有任何服务注册到 Eureka 中。

从图 3-6 中可以得到很多关于 Eureka 服务的有用信息，其中的部分信息实际上是可以通过配置手段进行调整和管理的。Eureka 作为注册中心的实现方案，运行时需要依赖诸多配置信息，下面就来梳理一下 Eureka 的常见配置。

在 eureka-server 工程的 application.yml 文件中，我们添加了如下配置信息。Eureka 服务相关的配置信息相对比较复杂，除了通用的服务端口设置还存在很多特定的配置项值得我们去进一步挖掘。

```
server:
  port: 8761
```

```
eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
```


Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

General Info

Name	Value
total-avail-memory	298mb
environment	test
num-of-cpus	4
current-memory-usage	79mb (26%)
server-uptime	00:01
registered-replicas	http://localhost:8761/eureka/
unavailable-replicas	http://localhost:8761/eureka/,
available-replicas	

Instance Info

Name	Value
ipAddr	192.168.0.103
status	UP

图 3-6　Eureka 服务监控页面

在 Eureka 中，配置信息可以分为以下 3 大类，我们要做的事情就是利用这些配置项来控制 Eureka 的具体行为。

- 服务端配置项

服务端配置项用于控制服务器端行为，以 eureka.server 开头。

- 客户端配置项

客户端配置项从客户端角度出发考虑配置需求，以 eureka.client 开头。

- 实例类配置项

实例类配置项则关注于注册到 Eureka 的服务实例本身，以 eureka.instance 开头。

以上三类配置项是从角色的维度出发进行区分，但实际上，一个服务本身可能具备多个角色。例如，对于 Eureka 服务而言，显然适用于服务端配置项；而作为一个微服务，它实际上也可以作为客户端注册到注册中心，这时候它使用的就是客户端配置项；一旦注册成功，它就成为注册中心中的一个实例，实例类配置项就可以用来控制其运行时行为。

在上述代码中，可以看到两个以 eureka.client 开头的客户端配置项，分别是 registerWith

Eureka 和 fetchRegistry。其中，registerWithEureka 用于指定该客户端实例是否在 Eureka 服务器上注册自己的信息以供其他服务进行发现；而 fetchRegistry 用于指定该客户端是否获取 Eureka 服务器上的注册信息。这两个配置项默认都是 true，但这里都将其设置为 false。因为在微服务体系中，包括 Eureka 服务在内的所有服务对于注册中心来说都可以算作是客户端，而 Eureka 服务显然不同于业务服务，我们不希望 Eureka 服务对自身进行注册。

3.2.2 构建 Eureka 服务器集群

上一节介绍了构建单个 Eureka 服务器的方法，这种运行 Eureka 服务的方式一般称为 Standalone 模式。考虑到单个 Eureka 服务可能存在的单点失效问题，我们通常都需要构建一个 Eureka 服务器集群。

我们可以通过 Eureka 服务互相注册的方式来实现高可用的部署，这种方式被称为 Peer Awareness 模式。而构建这种模式需要做的事情就是在配置文件中配置 defaultZone 为目标的注册中心即可。

现在准备两个 Eureka 服务实例 peer1 和 peer2，为了方便描述，分别采用 application-peer1.yml 和 application-peer2.yml 来命名这两个服务中的配置文件。其中 application-peer1.yml 配置文件的内容如下，具体代码可参考本书提供的案例资源。

```
spring:
  application:
    name: eureka-server

server:
  port: 8761

eureka:
  instance:
    hostname: peer1
  client
    serviceUrl
      defaultZone: http://peer2:8762/eureka/
```

而 application-peer2.yml 配置文件的内容如下。

```
spring:
  application:
    name: eureka-server

server:
  port: 8762
```

```
eureka:
  instance:
    hostname: peer2
  client
    serviceUrl
      defaultZone: http://peer1:8761/eureka/
```

从上述代码中可以看到 application-peer1.yml 和 application-peer2.yml 中的配置项完全一致，从内容上看也只是调整了端口和地址的引用。这里引入了一个新的客户端配置项 eureka.client.serviceUrl.defaultZone 用于指向集群中的其他 Eureka 服务器。所以 Eureka 集群的构建方式实际上就是将自己作为服务并向其他注册中心注册自己，这样就形成了一组互相注册的服务注册中心从而实现服务列表的同步。显然，这个场景下 registerWithEureka 和 fetchRegistry 配置项应该都使用其默认的 true 值。

如果我们使用本机搭建集群环境，显然 eureka.instance.hostname 配置项中的 peer1 和 peer2 是无法访问的，所以需要在本机 hosts 文件中添加以下信息。

```
127.0.0.1 peer1
127.0.0.1 peer2
```

启动这两个 Eureka 服务，分别打开 http://127.0.0.1:8761/和 http://127.0.0.1:8762/端点即可看到各自的服务注册效果。一方面，我们可以看到这两个服务之间实现了相互注册，并且在 registered-replicas 和 available-replicas 中都能看到可用的副本信息。图 3-7 展示了 peer1 中的 Eureka 集群信息，peer2 中的 Eureka 集群信息也与此类似。

DS Replicas

peer2

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
EUREKA-SERVER	n/a (2)	(2)	UP (2) - DESKTOP-IBI7JFP:eureka-server:8761 , DESKTOP-IBI7JFP:eureka-server:8762

General Info

Name	Value
total-avail-memory	761mb
environment	test
num-of-cpus	4
current-memory-usage	381mb (50%)
server-uptime	01:38
registered-replicas	http://peer2:8762/eureka/
unavailable-replicas	
available-replicas	http://peer2:8762/eureka/,

图 3-7　peer1 中的 Eureka 集群信息

另一方面，在两个 Eureka 服务的控制台日志中能够看到如下信息，表示 Eureka 服务器集群已经启动成功。

```
  AbstractInstanceRegistry: Registered instance EUREKA-SERVER/ DESKTOP-IBI7JFP:
eureka-server:8761 with status UP (replication=true)
  AbstractInstanceRegistry: Registered instance EUREKA-SERVER/ DESKTOP-IBI7JFP:
eureka-server:8762 with status UP (replication=true)
  PeerAwareInstanceRegistryImpl: Got 2 instances from neighboring DS node
  PeerAwareInstanceRegistryImpl: Renew threshold is: 3
  PeerAwareInstanceRegistryImpl: Changing status to UP
```

手动关闭 peer2 服务后，就会出现如图 3-8 所示的集群信息，可以看到集群中某个服务不可用的警告信息，同时展示具体的宕机服务。

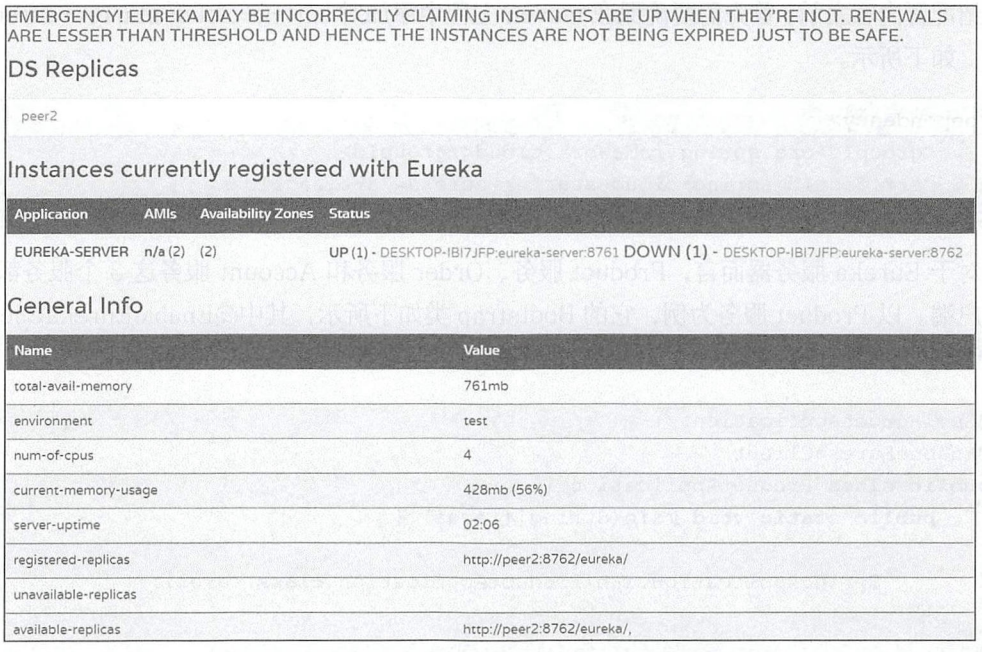


图 3-8　关闭 peer2 服务之后的 Eureka 集群信息

3.3　使用 Eureka 注册和发现服务

第 1 章介绍了贯穿全书的 OrderSystem 案例，并引出构成该案例的 Product 服务、Order 服务和 Account 服务等 3 个独立的微服务，它们的 maven 代码工程名分别为 product-service、order-service 和 account-service。这 3 个服务属于业务服务，本节将通过这 3 个服务来演示如何通过 Eureka 服务器实现服务的注册和发现。

注意：因为 Product 服务、Order 服务和 Account 服务这 3 个业务服务的代码中分别存在名为 ProductService、OrderService 和 AccountService 等 Service 层的实现类，为了避免混淆，对这 3 个业务服务不采用 product-service、order-service 和 account-service 这种代码工程名称来进行描述，而是统一使用 Product 服务、Order 服务和 Account 服务。而对于其他诸如注册中心服务、配置服务等非业务类的服务而言，我们将直接使用 eureka-server、config-server 这样的 maven 工程名称进行描述。

3.3.1 通过配置实现服务注册

使用 Eureka 注册一个基于 Spring Boot 的服务非常简单，主要工作也是通过配置来完成。在介绍配置内容之前，首先需要确保在 maven 工程中添加对 spring-cloud-starter-eureka 的依赖，如下所示。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

对于 Eureka 服务器而言，Product 服务、Order 服务和 Account 服务这 3 个服务都是它的客户端。以 Product 服务为例，它的 Bootstrap 类如下所示，其中@EnableEurekaClient 注解用于表明启动 Eureka 客户端，这样该服务就可以自动注册到 Eureka 服务器。

```
@SpringBootApplication
@EnableEurekaClient
public class ProductApplication {
    public static void main(String[] args) {

        SpringApplication.run(ProductApplication.class, args);
    }
}
```

接下来就是最重要的配置工作，Product 服务中的配置内容如下所示，我们重点关注与注册中心相关的配置项。可以看到，这里配置了两个角色的行为，一个作为 Eureka 服务器的客户端，一个作为注册到 Eureka 中的服务实例。

```
spring:
  application:
    name: productservice
  profiles:
    active:
      default
```

```
server:
  port: 8081

eureka:
  instance:
    preferIpAddress: true
  client:
    registerWithEureka: true
    fetchRegistry: true
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

在eureka.client段中，可以看到registerWithEureka、fetchRegistry和serviceUrl这3个已经介绍过的配置项。registerWithEureka和fetchRegistry的默认值是true，这里再次显式的指定是为了强调它们的特性；而serviceUrl.defaultZone指定的就是Eureka服务器的地址。

在eureka.instance段中，引入了一个新的配置项preferIpAddress用来指明使用IP而不是服务名称来注册自身服务。因为Eureka默认是使用域名进行服务的注册和解析，为了避免不必要的错误，建议这里直接使用IP。

除了eureka.client和eureka.instance段，这里还有一个配置项与注册中心直接相关，那就是服务的名称。在上面的示例中，Product服务的名称通过“spring.application.name=productservice”进行指定，也就是说Product服务在注册中心的名称为productservice。在后续的示例中，我们会使用这一名称获取Product服务在Eureka中的各种注册信息。

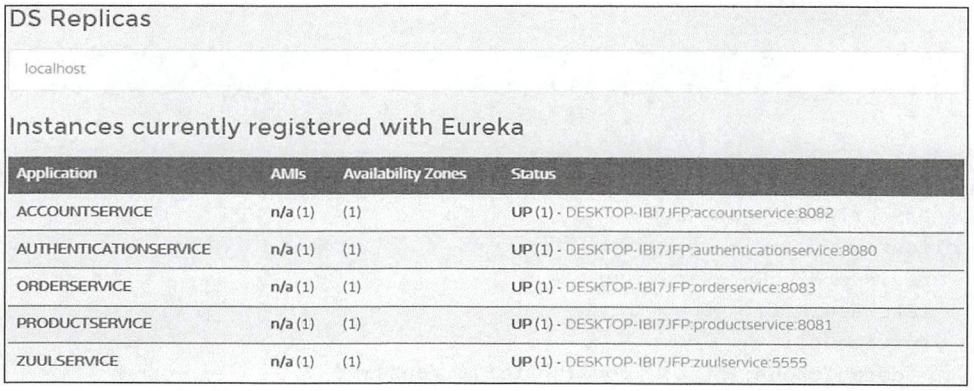
当然，如果已经构建了Eureka服务器集群，那么eureka.client.serviceUrl.defaultZone配置项的内容就应该是“http://peer1:8761/eureka/, http://peer2:8762/eureka/”，用于指向当前的集群环境。读者可自行将某个服务注册到Eureka服务器集群，然后尝试关闭其中一个Eureka服务的场景。

3.3.2 获取服务注册信息

当我们分别创建并启动Product服务、Order服务、Account服务以及后续章节会陆续介绍的一些其他非业务类服务（如用于服务认证的AuthenticationService和用于服务路由的ZuulService）时，Eureka服务器的当前状态如图3-9所示。

为了获取注册到Eureka服务器上具体某一个服务实例的详细信息，可以访问地址http://<eureka service>:8761/eureka/apps/<APPID>，该地址代表的就是一个普通的HTTP GET请求。以Product服务为例，发送HTTP请求到http://localhost:8761/eureka/apps/productservice，并把获取的结果转换为JSON格式，可以得到如下信息。

仅供非商业用途或交流学习使用


DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
ACCOUNTSERVICE	n/a (1)	(1)	UP (1) - DESKTOP-IBI7JFP:accountservice:8082
AUTHENTICATIONSERVICE	n/a (1)	(1)	UP (1) - DESKTOP-IBI7JFP:authenticationservice:8080
ORDERSERVICE	n/a (1)	(1)	UP (1) - DESKTOP-IBI7JFP:orderservice:8083
PRODUCTSERVICE	n/a (1)	(1)	UP (1) - DESKTOP-IBI7JFP:productservice:8081
ZUULSERVICE	n/a (1)	(1)	UP (1) - DESKTOP-IBI7JFP:zuulservice:5555

图 3-9　包含若干服务信息的 Eureka 服务监控页面

```
{
    "application": {
        "name": "PRODUCTSERVICE",
        "instance": [
            {
                "instanceId": "DESKTOP-IBI7JFP:productservice:8081",
                "hostName": "10.20.0.12",
                "app": "PRODUCTSERVICE",
                "ipAddr": "10.20.0.12",
                "status": "UP",
                "overriddenstatus": "UNKNOWN",
                "port": {
                    "$": 8081,
                    "@enabled": "true"
                },
                "securePort": {
                    "$": 443,
                    "@enabled": "false"
                },
                "countryId": 1,
                "dataCenterInfo": {
                    "@class": "com.netflix.appinfo.InstanceInfo$DefaultData
                    CenterInfo",
                    "name": "MyOwn"
                },
                "leaseInfo": {
                    "renewalIntervalInSecs": 30,
                    "durationInSecs": 90,
                    "registrationTimestamp": 1518328374945,
                    "lastRenewalTimestamp": 1518328583368,
                    "evictionTimestamp": 0,
```

仅供非商业用途或交流学习使用

```
                    "serviceUpTimestamp": 1518328373925
                },
                "metadata": {
                    "@class": "java.util.Collections$EmptyMap"
                },
                "homePageUrl": "http://10.20.0.12:8081/",
                "statusPageUrl": "http://10.20.0.12:8081/info",
                "healthCheckUrl": "http://10.20.0.12:8081/health",
                "vipAddress": "productservice",
                "secureVipAddress": "productservice",
                "isCoordinatingDiscoveryServer": "false",
                "lastUpdatedTimestamp": "1518328374947",
                "lastDirtyTimestamp": "1518328373113",
                "actionType": "ADDED"
            }
        ]
    }
}
```


根据如上所示的 Product 服务实例详细信息，可以获取该服务的服务名称、IP 地址、端口、是否可用等基本信息，也可以访问 statusPageUrl、healthCheckUrl 等地址查看当前服务的运行状态，更为重要的是得到了 leaseInfo、actionType 等与服务注册过程直接相关的基础数据，这些基础数据有助于我们理解 Eureka 作为注册中心的工作原理。

3.4 Eureka 基本架构

通过前面几节内容的介绍，我们看到 Eureka 服务器的构建和使用都非常简单。对于这类工具，我们更应该剖析其内部构造。本节将深入分析 Eureka 的基本架构，以便读者能够掌握设计和实现注册中心的基本思路和方法。

3.4.1 Eureka 服务注册和发现架构

Eureka 是由 Netflix 开源的一款提供服务注册和发现的产品，其基本架构如图 3-10 所示。从图中可以看到该架构由 Eureka 服务器、服务提供者和服务消费者这 3 个逻辑角色所组成，并与 3.1.1 节中介绍的服务注册中心模型具有高度一致性。

对图 3-10 中的基本架构进一步展开，可以得到如图 3-11 所示的细化架构，该图进一步展示了 3 个角色之间的交互：首先服务提供者会向 Eureka 服务器提交服务管理相关操作；而 Eureka 服务器之间会做注册服务的同步从而保证状态一致；同时，服务消费者会向 Eureka 服

仅供非商业用途或交流学习使用

务器获取注册服务列表并消费服务。

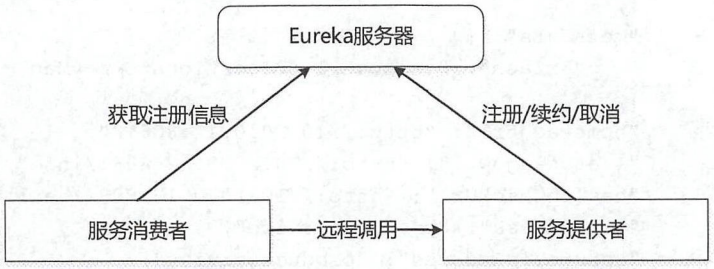


图 3-10　Netflix Eureka 基本架构

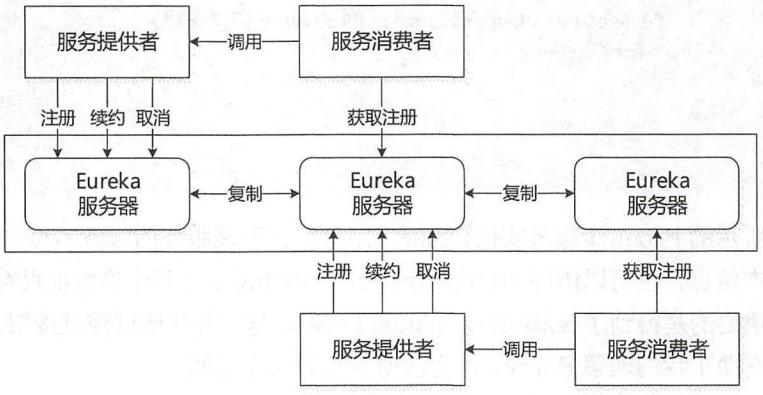


图 3-11　Netflix Eureka 细化架构

在图 3-11 中，Eureka 有以下几个概念与服务治理直接相关。

- 服务注册（Register）

当 Eureka 客户端向 Eureka 服务器注册时，它提供自身的元数据，比如 IP 地址、端口等信息。

- 服务续约（Renew）

在服务续约中，Eureka 客户端会每隔 30 秒发送一次心跳来进行服务续约。通过续约来告知 Eureka 服务器该客户端仍然存在，希望服务器不要剔除自己。

- 服务下线（Cancel）

Eureka 客户端在程序关闭时向 Eureka 服务器发送取消请求。发送请求后，该客户端实例信息将从服务器的实例注册列表中删除。

- 服务剔除（Eviction）

在默认情况下，当 Eureka 客户端连续 90 秒没有向 Eureka 服务器发送服务续约的心跳（Heartbeat），Eureka 服务器就会将该服务实例从服务注册列表中删除，即剔除该服务。

- 获取服务注册列表信息（Fetch Registries）

仅供非商业用途或交流学习使用

Eureka 客户端从 Eureka 服务器获取服务注册列表信息，并将其缓存在本地。客户端会使用该信息查找其他服务，从而进行远程调用。该注册列表信息定期（每 30 秒）更新一次。每次返回的注册列表信息可能与 Eureka 客户端的缓存信息有所不同，Eureka 客户端会自动处理两者之间的差异。

显然，对于一个注册中心而言，想要完整理解全局性的设计理念和原理，需要分别从注册中心服务器、服务提供者、服务消费者这 3 个角色出发，分析它们之间的交互细节。

（1）注册中心服务器

想要实现服务管理，首先必须保存服务的定义，在 Eureka 中使用两层 HashMap 来达到服务存储的效果。第一层 HashMap 的 Key 是 App Name，也就是应用名字，而第二层 HashMap 的 Key 是 Instance Name，也就是实例名字。

面向服务注册中心的操作主要包括服务注册、服务续约和服务下线。实际上这 3 种不同的操作对于注册中心服务器而言其执行的工作流程基本一致，它们都是对服务存储的操作，并把这一操作同步到其他 Eureka 节点。以服务注册为例，图 3-12 展示了主要组件之间的交互过程及核心操作。服务续约是一个定时操作，类似于心跳。Eureka 会定期（默认为每 60 秒）检测失效的服务，检测标准就是看服务在一定时间内有没有成功续约。而服务下线用来把自身的服务从 Eureka 服务器中删除，以免客户端调用不存在的服务。

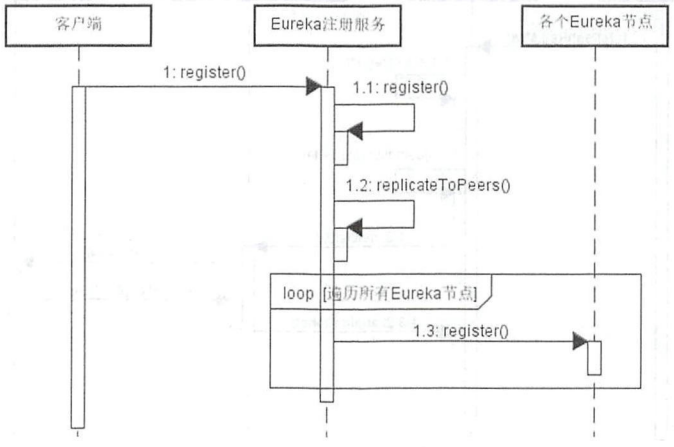


图 3-12　Netflix Eureka 服务注册时序图

注册中心服务器的另一个核心功能是提供服务列表，为了提高性能，服务列表在 Eureka 服务器会缓存一份，同时每 30 秒更新一次。

（2）服务提供者

服务提供者关注服务注册、服务续约和服务下线等功能，对于注册中心服务器而言，服务提供者相当于客户端，所以它可以使用服务器提供的 RESTful API（基于 Jersey 框架实现）完成上述操作。下面以基于心跳操作的服务续约为例，给出服务提供者的大致交互流程，如

仅供非商业用途或交流学习使用

图 3-13 所示。

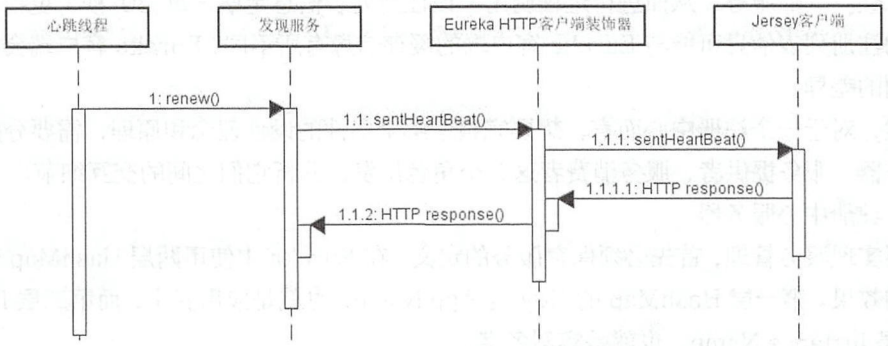


图 3-13 服务提供者服务续约时序图

（3）服务消费者

服务消费者与服务提供者本质上是同一类客户端。如同 3.1.1 节中介绍的注册中心实现方案，服务消费者在本地会有一份服务列表信息的缓存，所以需要定期更新缓存，更新操作的流程图如图 3-14 所示。

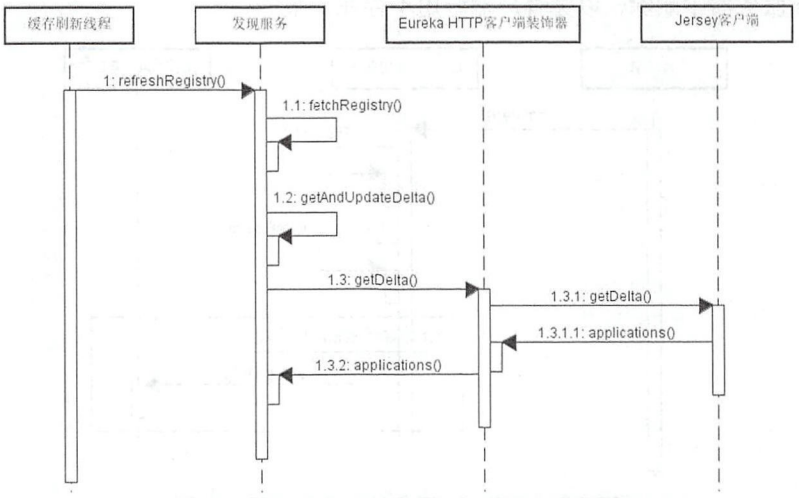


图 3-14 服务消费者定期更新服务缓存时序图

3.4.2 Eureka 高可用架构

服务注册中心虽然是一个侧重于服务治理的辅助性组件，同样也需要确保可用性。Eureka 服务器的高可用，实际上就是将自己作为服务向其他服务注册中心注册自己，这样就形成了一组互相注册的服务注册中心以实现服务列表的互相同步，达到高可用的效果。

仅供非商业用途或交流学习使用

在图 3-12 中可以看到一个非常重要的 replicateToPeers 操作，该操作就是用来实现服务器节点之间的状态同步。通过这种方式，任意一个 Eureka 服务器获取来自客户端的通知之后就能够保证状态会在所有的 Eureka 服务器中得到更新。具体实现方式其实很简单，就是接收到请求的 Eureka 服务器把请求再次转发给其他的 Eureka 服务器，调用同样的接口，传入同样的参数，唯一不同之处在于会告诉其他 Eureka 服务器不需要再进行数据复制。

剩下的一个问题就是如何知道有哪些 Eureka 服务器节点。Eureka 服务器在启动后会调用 EurekaClientConfig.getEurekaServerServiceUrls 来获取所有的 Peer 节点，并且会定期更新（见图 3-15）。这个方法的默认实现是从配置文件读取服务器地址，所以如果 Eureka 服务器节点相对固定的话，可以通过在配置文件中配置节点列表的方式来实现。如果在运行过程中新加入了一个节点或者服务器重启，那么每个节点都可以把自己当作是服务消费者，并从其他 Eureka 服务器节点获取所有服务的注册信息。然后再对每个服务在自己这里执行注册操作，从而完成初始化。

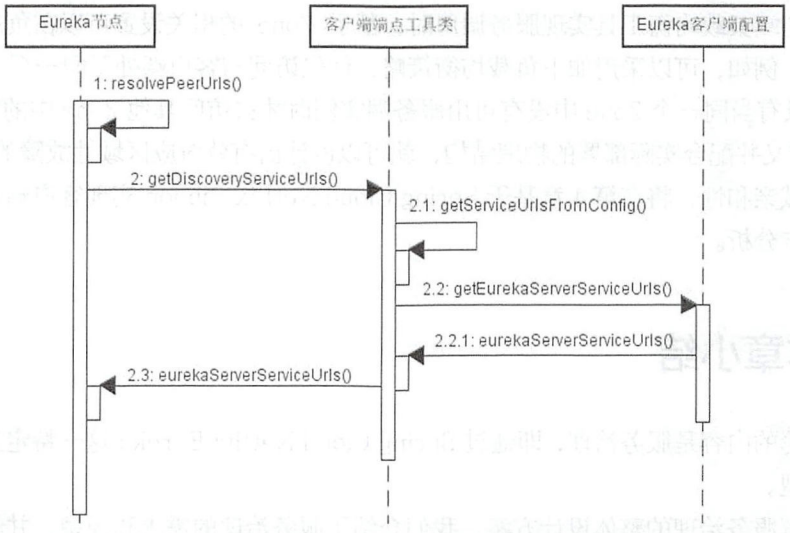


图 3-15　Netflix Eureka 获取服务器节点时序图

3.4.3　Eureka 区域亲和性

在 3.2 节介绍了 eureka.client.serviceUrl.defaultZone 配置项，这里的“Zone”表示的是区域的概念。在 Eureka 中除了 Zone 还存在 Region，我们可以简单地把 Region 理解为 Eureka 集群，Zone 理解成机房。结合图 3-16 所示的结构图，可以看到一个 Eureka 集群被部署在了 Zone1 和 Zone2 两个机房中。

从 Region 和 Zone 的关系我们可以得出区域亲和性的概念。所谓区域亲和性（Zone

仅供非商业用途或交流学习使用

Affinity）是指服务消费者在选择服务时会选择与该服务消费者在同一个 Zone 的服务，这样做的目的是为了减少服务调用的延迟。当我们没有专门为 Region 配置 Zone 时，Eureka 将默认采用 defaultZone，这也是前面介绍 eureka.client.serviceUrl.defaultZone 配置项的由来。若要为应用程序指定 Zone，可以通过 eureka.client.availabilityZones 属性来进行设置，示例配置如下所示。通过 Region 可以获得一个 Zone 的列表，所以 Region 和 Zone 是一对多的关系。

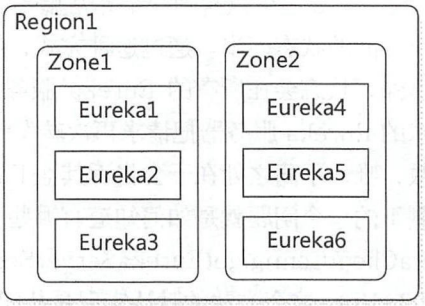


图 3-16　Netflix Eureka 中的 Region 和 Zone

```
eureka.client.region=myregion
eureka.client.availabilityZones.myregion=myzone1, myzone2
```

使用客户端负载均衡工具实现服务调用时，基于 Zone 的相关设置可以在负载均衡时提供区域亲和性。例如，可以采用如下负载均衡策略：优先访问与客户端处于同一个 Zone 中的服务端实例，只有当同一个 Zone 中没有可用服务端实例时才去访问其他 Zone 中的实例。所以通过 Zone 的定义并配合实际部署的物理结构，就可以设计出有效对应区域性故障的容错集群。

关于区域亲和性，将在第 4 章基于 Spring Cloud Netflix Ribbon 实现客户端负载均衡的场景中做进一步分析。

3.5　本章小结

本章讨论的内容是服务治理，即通过 Spring Cloud Netflix Eureka 这一特定工具实现服务注册中心模型。

为了理解服务治理的整体设计方案，我们介绍了服务治理的需求和模型，并给出了服务治理的基本方案。以注册中心为代表的技术实现机制被广泛应用于服务治理解决方案的设计过程中，业界也存在一批优秀的工具用来实现注册中心，显然不同的工具在具备共性的同时也各有特色，我们重点对这些工具的共性进行了分析和总结。

Eureka 是 Spring Cloud 中注册中心的实现框架，我们需要从以下两个问题出发看待 Eureka：一个是如何构建 Eureka 服务器；另一个是如何使用 Eureka 注册和发现服务。本章对这两个问题给出了答案，并从 Eureka 基本架构的角度出发讨论了它的服务注册和发现机制、高可用架构和区域亲和性。

Eureka 服务本身也是一个微服务，其他业务服务需要注册到 Eureka 服务中完成服务定义的共享以便实现负载均衡。在下一章中，我们将基于注册中心来实现客户端负载均衡并提供相关工具。

仅供非商业用途或交流学习使用

第 4 章

Spring Cloud Netflix Ribbon 与负载均衡

在上一章中，我们已经知道所有的服务定义都存放在注册中心 Eureka 服务器中。当能够从 Eureka 服务器获取某一个服务的各个运行实例信息时，原则上我们就可以执行负载均衡策略。Spring Cloud 中存在一个组件专门用来实现负载均衡，这就是本章要介绍的 Spring Cloud Netflix Ribbon。

基于 Spring Cloud Netflix Ribbon 实现负载均衡的结构图如图 4-1 所示，可以看到各个微服务实例会和 Ribbon 服务一样注册到 Eureka 服务器，然后 Ribbon 从 Eureka 服务器中获取所有注册服务的服务列表。一旦获取服务列表，Ribbon 就能通过各种负载均衡策略实现服务调用。

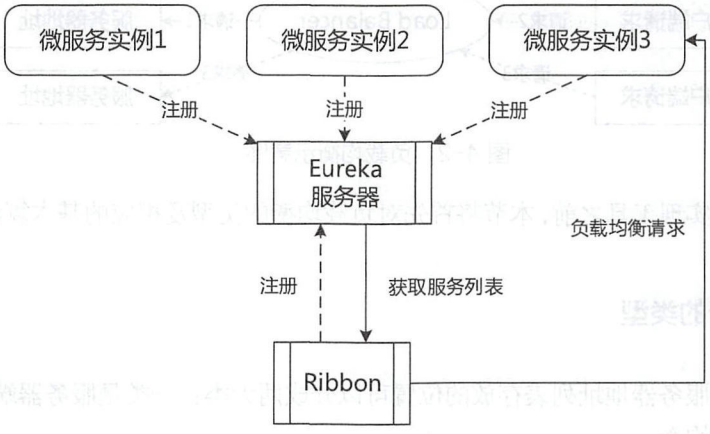


图 4-1　使用 Ribbon 实现负载均衡结构图

根据服务列表的获取和存储方式，负载均衡可以分为客户端负载均衡和服务器端负载均衡两种形式，图 4-1 中展示的是一种客户端负载均衡机制。Ribbon 作为一种独立的客户端负载均衡实现工具，一方面可以通过 Eureka 完成负载均衡，另一方面它也支持直接访问服务列表。本章将首先介绍负载均衡的基本概念，然后介绍使用 Ribbon 实现客户端负载均衡的方法，最后对 Ribbon 基本架构进行展开。

仅供非商业用途或交流学习使用

4.1 负载均衡

分布式系统诞生的背景在于单机处理能力存在瓶颈，而升级单机处理能力的性价比越来越低，同时系统的稳定性和可用性也日益受到重视。从横向拆分的角度讲，分布式是指将不同的业务分布在不同的地方，而集群（Cluster）指的是将几台服务器集中在一起，实现同一业务。分布式中的每一个节点，都可以做集群，但集群并不一定就是分布式。集群概念的提出考虑到了分布式系统中的性能问题，即集群能够将业务请求分摊到多台单机性能不一定非常出众的服务器，这就是本章将要介绍的负载均衡机制。

所谓负载均衡（Load Balance，LB），简单地讲就是将请求分摊到多个操作单元上进行执行，图 4-2 中来自客户端的请求通过负载均衡机制将被分发到各个服务器，根据分发策略的不同将产生不同的分发结果。负载均衡在实现上可以使用硬件、软件，或者两者兼有，本书主要介绍基于软件的负载均衡机制。

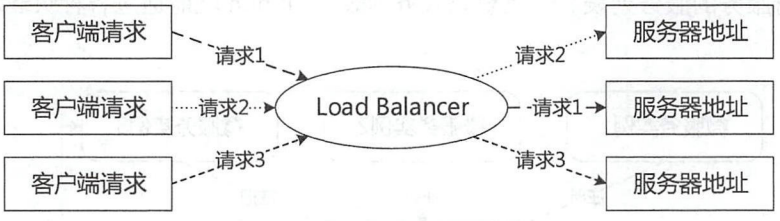


图 4-2 负载均衡示意图

在引入具体的实现工具之前，本节将首先对负载均衡的类型及相应的基本算法做简要介绍。

4.1.1 负载均衡的类型

负载均衡根据服务器地址列表存放的位置可以分成两大类：一类是服务器端负载均衡；另一类是客户端负载均衡。

1. 服务器端负载均衡

在微服务架构中，图 4-3 展示了服务器端负载均衡机制示意图，首先客户端发送请求到负载均衡器，然后这台负载均衡器负责将接收到的各个请求转发到运行中的某个微服务节点上，最后接收到请求的微服务节点做响应处理。提供服务器端负载均衡的工具有很多，例如常见的 Apache、Nginx、HAProxy 等都实现了基于 HTTP 协议或 TCP 协议的负载均衡模块。

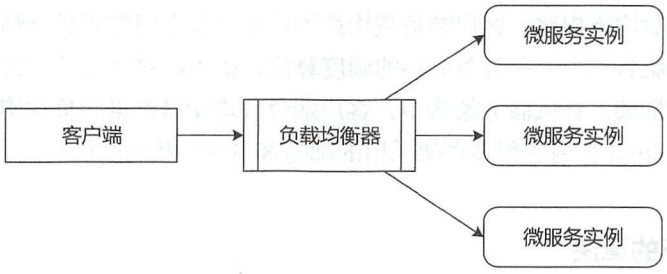


图 4-3　服务器端负载均衡示意图

基于服务器端的负载均衡机制实现比较简单，只需要在客户端与各个微服务实例之间架设集中式的负载均衡器即可。负载均衡器与各个微服务实例之间需要实现服务诊断以及状态监控，通过动态获取各个微服务的运行时信息决定负载均衡的目标服务。如果负载均衡器检测到某个服务不可用时就会自动移除该服务。

通过上述分析，可以看到负载均衡器运行在一台独立的服务器上并充当代理（Proxy）的作用。所有的请求都需要通过负载均衡器的转发才能实现服务调用，这可能会是一个问题，因为当服务请求量越来越大时，负载均衡器将会成为系统的瓶颈。同时，一旦负载均衡器自身发生失败，整个微服务的调用过程都将发生失败。因此，为了避免集中式负载均衡所带来的这种问题，客户端负载均衡同样也是一种常用的方式。

2. 客户端负载均衡

客户端本身同样可以实现负载均衡，客户端负载均衡最基本的表现形式如图 4-4 所示。客户端负载均衡机制的主要优势在于不会出现集中式负载均衡所产生的瓶颈问题，因为每个客户端都有自己的负载均衡器，该负载均衡器的失败不会造成严重的后果。但是，由于所有运行时信息都需要在多个负载均衡器之间进行传递，会在一定程度上加重网络流量负载。

客户端负载均衡，简单地说就是在客户端程序内部设定一个调度算法，在向服务器发起请求时，先执行调度算法计算出目标服务器地址。在图 4-4 所示的客户端中包含着各个微服务实例信息，然后通过负载均衡算法计算目标服务器实现负载均衡。

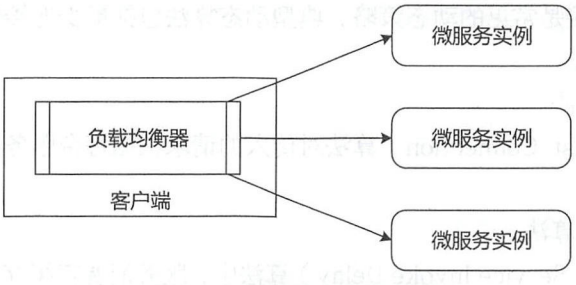


图 4-4　客户端负载均衡示意图

与服务器端负载均衡相比，客户端负载均衡不需要架设专门的服务器端代理。如果客户端应用程序能够获取服务列表，并具备成熟的调度算法，就可以对外提供相关的 API，从而成为一种独立的工具和框架。在微服务架构中，客户端负载均衡是常见的负载均衡实现方案，包括 Dubbo 和 Spring Cloud 在内的很多框架采用的都是客户端负载均衡机制。

4.1.2 负载均衡的算法

无论是使用服务器端负载均衡还是客户端负载均衡，运行时的分发策略决定了负载均衡的效果。分发策略在软件负载均衡中的实现形式为一组分发算法，通常称为负载均衡算法。负载均衡算法可以分成两大类，即静态负载均衡算法和动态负载均衡算法。

1. 静态负载均衡算法

静态负载均衡算法的代表是随机（Random）算法和轮询（Round Robin）算法。

- 随机算法

在集群中采用随机算法进行负载均衡的结果相对比较平均。随机算法的实现也比较简单，使用 JDK 自带的 Random 随机算法就可以指定服务提供者的地址。随机算法的一种改进是加权随机（Weight Random）算法，在集群中可能存在部分性能较优的服务器，为了使这些服务器响应更多请求，可以通过加权随机算法提升这些服务器的权重。

- 轮询算法

加权轮询（Weighted Round Robin）算法同样使用权重，一般的流程为顺序循环遍历服务提供者列表，到达上限之后重新归零，继续顺序循环直到指定某一台服务器作为服务的提供者。普通的轮询算法实际上就是权重为 1 的加权轮询算法。

2. 动态负载均衡算法

所有涉及权重的静态算法都可以转变为动态算法，因为权重可以在运行过程中动态更新。例如，动态轮询算法中权重值基于对各个服务器的持续监控并不断更新。另外，基于服务器的实时性能分析分配连接是常见的动态策略。典型动态算法包括最少连接数算法、服务调用时延算法和源 IP 哈希算法。

- 最少连接数算法

最少连接数（Least Connection）算法对传入的请求根据每台服务器当前所打开的连接数来分配。

- 服务调用时延算法

在服务调用时延（Service Invoke Delay）算法中，服务消费者缓存并计算所有服务提供者的服务调用时延，根据服务调用和平均时延的差值动态调整权重。

- 源 IP 哈希算法

源 IP 哈希（Source IP Hash）算法实现请求 IP 粘滞（Sticky）连接，尽可能让消费者总是向同一提供者发起调用服务。这是一种有状态机制，也可以归为动态负载均衡算法。

4.2 使用 Ribbon 实现客户端负载均衡

Spring Cloud Netflix Ribbon 是一种客户端负载均衡方案，本节将首先对 Spring Cloud Netflix Ribbon 做简要介绍，并给出使用 Spring Cloud Netflix Ribbon 进行服务调用的具体实现方法和相关示例。

4.2.1 Spring Cloud Netflix Ribbon 简介

Netflix Ribbon 是 Netflix 开源的一款用于客户端软负载均衡的工具软件，主要功能是提供客户端的软件负载均衡算法。简单地说就是在配置文件中列出负载均衡器背后所有服务实例列表，Netflix Ribbon 会自动地基于某种负载均衡算法（如随机、轮询等）去连接服务实例。我们也很容易使用 Netflix Ribbon 实现自定义的负载均衡算法。同时，Netflix Ribbon 客户端组件提供了一系列完善的诸如连接超时和重试等辅助机制。

因为 Netflix Ribbon 本质上只是一个工具，而不是一套完整的解决方案，所以 Spring Cloud Netflix Ribbon 对 Netflix Ribbon 做了封装以便更好地使用 Spring Boot 的自动配置理念。通过 Spring Cloud Netflix Ribbon 大大简化了在微服务架构中使用客户端负载均衡的成本，通过注解就能简单实现在面向服务的接口调用中自动集成负载均衡功能。为了简化描述，下文中使用 Ribbon 代表 Spring Cloud Netflix Ribbon。

Ribbon 不像服务注册中心 Eureka 那样需要独立部署，而是直接嵌入在服务消费者的代码中。基于 Spring Cloud Netflix Ribbon，有如下两种方式实现客户端负载均衡功能。

- 使用@LoadBalanced 注解和 RestTemplate

服务消费者直接通过使用被@LoadBalanced 注解修饰过的 RestTemplate 就可以自动实现服务调用过程中的负载均衡功能。

- 使用@RibbonClient 注解

Ribbon 还允许声明额外的配置项以完全控制客户端负载均衡行为，可以通过使用@RibbonClient 注解来实现更细粒度的客户端配置。

本节首先对基于@LoadBalanced 注解和 RestTemplate 的这种使用方法进行详细展开，关于@RibbonClient 注解的使用方法将在下节中进行展开。无论使用哪种方法，首先需要明确如何通过 Eureka 提供的 DiscoveryClient 工具类查找注册在 Eureka 的服务，这是 Ribbon 实现

仅供非商业用途或交流学习使用

客户端负载均衡的前提。

4.2.2 使用 DiscoveryClient 查找服务

在第 3 章中，已经有几个服务注册到 Eureka 了，通过 Eureka 提供的管理界面和 HTTP 端点可以获取服务的详细信息，但是这还不够。在现实应用中，很多时候我们希望通过代码在运行时能够实时获取注册中心中的服务列表，并通过服务定义动态发起服务调用（见图 4-5）。

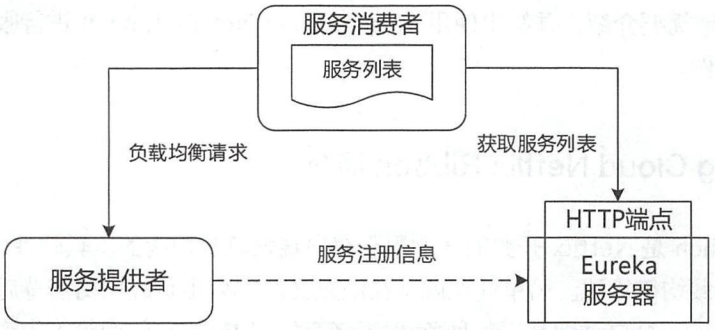


图 4-5 通过服务注册信息进行服务调用示意图

图 4-5 所示的通过服务注册信息进行服务调用的过程实际上就是客户端负载均衡的基本实现策略。如果不借助于 Ribbon 之类的客户端负载均衡框架，为了实现图 4-5 中的效果，可以引入 Eureka 提供的 DiscoveryClient 工具类。以下代码位于 eureka-server 工程中，展示了 DiscoveryClient 类的一些基本用法，该示例用于获取当前注册到 Eureka 中的所有服务实例并返回它们的 URL 信息。

```
@Autowired
private DiscoveryClient discoveryClient;

public List<String> getEurekaServices(){
    List<String> services = new ArrayList<String>();
    List<String> serviceNames = discoveryClient.getServices();

    for(String serviceName : serviceNames) {
        List<ServiceInstance> serviceInstances =
            discoveryClient.getInstances(serviceName);

        for(ServiceInstance instance :serviceInstances) {
            serviceNames.add(String.format("%s:%s",
                serviceName,instance.getUri()));
        }
```

仅供非商业用途或交流学习使用


```
        }

        return services;
    }
```

如果能够获取注册中心中的服务定义，就可以通过 ServiceInstance 对该服务进行调用。在Spring中存在一个功能非常强大的基于RESTful风格的HTTP调用工具类RestTemplate，通过该工具类可以使用 URL 轻松实现 HTTP 请求。下面以 Product 服务为例，添加一个ProductController并定义根据商品编号（ProductCode）获取商品的端点信息，代码如下所示。

```
@RestController
@RequestMapping(value="v1/products")
public class ProductController {

    @Autowired
    ProductService productService;

    @RequestMapping(value = "/{productCode}", method =
        RequestMethod.GET)
    public Product getProduct(@PathVariable String productCode) {

        Product product =
            productService.getProductByCode(productCode);
        return product;
    }
}
```

重新启动 Product 服务并注册到 Eureka，因为该服务的服务名称为“productservice”，所以可以通过以下代码（位于product-service工程中）对“v1/products”端点发起调用，这里用到了RestTemplate的exchange()方法封装整个HTTP请求调用过程。

```
@Autowired
RestTemplate restTemplate;

@Autowired
private DiscoveryClient discoveryClient;

public Product getProduct(String productCode) {
    List<ServiceInstance> instances =
        discoveryClient.getInstances("productservice");

    if (instances.size()==0)
     return null;
```

仅供非商业用途或交流学习使用


```
        String serviceUri
            String.format("%s/v1/products/%s",instances.get(0).getUri()
                .toString(), productCode);

        ResponseEntity<Product> result =
            restTemplate.exchange(serviceUri, HttpMethod.GET, null,
                Product.class, productCode);

        return result.getBody();
    }
```

在上述代码中，通过 instances.get(0)方法获取的是服务列表中的第一个服务，如果这里使用 4.1.2 节介绍的负载均衡算法，就相当于实现了一个自定义的客户端负载均衡工具。

在 Spring Cloud 中对注册在 Eureka 中的服务发起调用的方式还有很多，例如，接下来要介绍的客户端负载均衡的实现工具 Spring Cloud Netflix Ribbon 及声明式调用框架 Spring Cloud Netflix Feign，这些工具本质上也是依赖于 DiscoveryClient 实现的服务调用。

4.2.3 通过 RestTemplate 调用服务

本节同样使用 Product 服务进行演示，因为涉及负载均衡，所以先要创建两个 Product 服务实例。为了显示负载均衡环境下的调用结果，我们在 ProductController 中添加日志以方便在运行时观察控制台输出信息。ProductController 的代码如下所示。

```
@RestController
@RequestMapping(value="v1/products")
public class ProductController {

    private static final Logger logger =
        LoggerFactory.getLogger(ProductController.class);

    @Autowired
    ProductService productService;

    @Autowired
    private HttpServletRequest request;

    @RequestMapping(value = "/{productCode}", method =
        RequestMethod.GET)
    public Product getProduct(@PathVariable String productCode) {

        logger.info("Get product by code: {} from port: {}", productCode,
            request.getServerPort());
```

仅供非商业用途或交流学习使用


```
        Product product = productService.getProductByCode(productCode);
        return product;
    }
}
```

下面分别用 8081 和 8082 来设置这两个 Product 服务实例的端口并启动这两个服务，在 Eureka 中将得到如图 4-6 所示的效果，可以看到这两个服务实例都已经注册成功。

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
PRODUCTSERVICE	n/a (2)	(2)	UP (2) - DESKTOP-IBI7JFP:productservice:8081 , DESKTOP-IBI7JFP:productservice:8082

图 4-6　Eureka 中的两个 Product 服务实例信息

根据第 1 章给出的案例描述，Order 服务会对 Product 服务进行服务调用，所以接下来我们将视角转向 Order 服务。对于 Product 服务而言，Order 服务就是它的客户端，为了在 Order 服务中使用 Ribbon 提供的@LoadBalanced 注解，需要在 order-service 工程中导入 spring-boot-starter-ribbon 依赖，代码如下所示。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-ribbon</artifactId>
</dependency>
```

接着修改 Order 服务中的 Bootstrap 类 OrderApplication，通过@LoadBalanced 注解创建 RestTemplate。修改后的 OrderApplication 类代码如下所示。

```
@SpringBootApplication
@EnableEurekaClient
public class OrderApplication {

    @LoadBalanced
    @Bean
    public RestTemplate getRestTemplate(){
        return new RestTemplate();
    }

    public static void main(String[] args) {
```

仅供非商业用途或交流学习使用


```
        SpringApplication.run(OrderApplication.class, args);
    }
}
```

对于 Order 服务而言准备工作已经就绪，现在就可以编写访问 Product 服务的业务代码。我们在 order-service 工程中添加一个新的 ProductClient 类并加入以下代码，注意，http://productservice/v1/products/{productCode}中的“productservice”是在 Product 服务中配置的服务名称。

```
@Autowired
RestTemplate restTemplate;

public Product getProduct(String productCode) {
    ResponseEntity<Product> result =
        restTemplate.exchange("http://productservice/v1/products
          /{productCode}", HttpMethod.GET, null, Product.class, productCode);

    return result.getBody();
}
```

可以看到以上代码与 4.2.2 节中的代码在风格上非常相近，首先注入 RestTemplate，然后通过 RestTemplate 的 exchange()方法对 Product 服务进行远程调用。但是请注意，这里的 RestTemplate 已经具备了客户端负载均衡功能，因为我们在 OrderApplication 类中创建该 RestTemplate 时添加了@LoadBalanced 注解。

为了验证客户端负载均衡功能是否已经生效，可在 order-service 工程中添加一个 OrderController 用作服务访问的入口。同时再添加一个 OrderService 类用作处理业务逻辑，该 OrderService 类会通过 ProductClient 类获取商品信息并完成订单相关处理。OrderService 类的部分代码如下所示。

```
@Autowired
private OrderRepository orderRepository;

@Autowired
private ProductClient productClient;

public Order addOrder(String productCode) {

    Order order = new Order();
    order.setId(1L);

    Product product = productClient.getProduct(productCode);
    if(product == null) {
```

仅供非商业用途或交流学习使用


```
            return order;
        }

        order.setItem(product.getProductName());
        order.setCreateTime(new Date());

        orderRepository.save(order);

        return order;
    }
```

OrderController 类的端点定义如下所示，可以看到这里直接使用 OrderService 作为订单处理的入口。

```
@RestController
@RequestMapping(value="v1/orders")
public class OrderController {

     @Autowired
     private OrderService orderService;

    @RequestMapping(value = "/{productCode}", method =
        RequestMethod.POST)
    public Order saveOrder( @PathVariable("productCode") String
        productCode) {

        Order order = orderService.addOrder(productCode);
        return order;
    }

    ...
}
```

图 4-7 展示了通过 Postman 访问 OrderController 端点的界面效果，表示整个流程中 Order 服务器成功访问了 Product 服务。

如果多次发送该请求，可在端口为 8081 的 Product 服务中看到如下日志。

```
   INFO [productservice,,] 6148 --- [nio-8081-exec-5] c.t.p.controllers. Product
Controller : Get product by code: 001 from port: 8081
```

同样，在端口为 8082 的 Product 服务中也可以看到类似服务请求的日志。

```
   INFO [productservice,,] 6148 --- [nio-8081-exec-5] c.t.p.controllers. Product
Controller : Get product by code: 001 from port: 8082
```

仅供非商业用途或交流学习使用

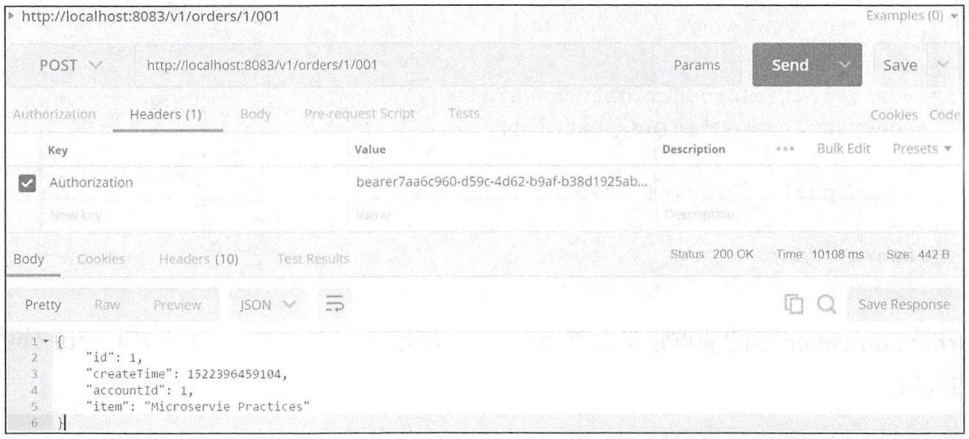


图 4-7　通过 Postman 访问 OrderController 端点示意图

从以上控制台日志中的时间信息可以看到这两个 Product 服务实例在交替的响应请求，意味着客户端负载均衡已经发挥了效果。

现在关闭两个 Product 服务实例中的任意一个（如端口号为 8081 的服务），然后再通过 Postman 进行服务调用，可以看到每次请求都是落在了端口号为 8082 的 Product 服务上。

在案例 OrderSystem 中还存在一个 Account 服务，Order 服务同样也会调用 Account 服务中的相关端点，所采用的是完全一样的负载均衡策略，读者可参考本书提供的完整的案例代码，这里不再赘述。

4.3　Ribbon 基本架构

Spring Cloud Netflix Ribbon 在使用方式上比较简单，但是当我们想要对它的运行过程进行控制时就需要理解其基本架构，并掌握控制其运行结果的相关概念和配置内容。Spring Cloud Netflix Ribbon 是一种辅助性的工具组件，在 Spring Cloud 整个体系中应用十分广泛，在本书第 6 章讲到 API 网关时还会再次介绍 Ribbon 的不同用法。

我们已经在上一节中看到，通过 Spring Cloud 对 Netflix Ribbon 的封装，可以将基于 Spring 自带 RestTemplate 发出的请求自动转换为客户端负载均衡的服务调用。本节将先从 Netflix Ribbon 基本架构讲起，并讨论常见的负载均衡策略如何在该框架中得以实现，之后还会简要介绍 Spring Cloud 对 Netflix Ribbon 的封装功能，并提供使用@RibbonClient 注解对服务调用进行精细化控制的实现方法。

仅供非商业用途或交流学习使用

4.3.1 Ribbon 核心机制

作为客户端负载均衡，要做的事情无非就是以下两件：

- 服务器列表中有哪些服务
- 如何从这些服务中选择一个服务进行调用

Netflix Ribbon 的核心接口 ILoadBalancer 就是围绕这两个问题来设计的，包括了如下基本方法。

- addServers(List<Server> newServers)

加载后端服务。

- chooseServer(Object key)

选择一个后端服务。

- markServerDown(Server server)

标记一个服务不可用。

- getAllServers()

获取所有后端服务列表。

- getReachableServers()

获取当前可用的服务列表。

ILoadBalancer 接口最基本的实现类是 BaseLoadBalancer，可以说负载均衡的核心功能都在这个类中得以实现，而该类同样包含了作为一个负载均衡器应该具备的一些核心组件。

- IRule

负载均衡策略，可以插件化地为负载均衡提供各种适用的负载均衡算法。我们在下一节中会对各种负载均衡算法进行具体展开。

- IPing

判断目标服务是否存活。IPing 接口中只有一个 isAlive()方法，通过对服务发出“IPing”操作来获取服务响应，从而判断该服务是否可用。IPing 的实现策略也有很多，包括 PingUrl、PingConstant、NoOpPing 和 DummyPing 等。

- LoadBalancerStats

记录负载均衡的实时运行信息，用来作为负载均衡策略的运行时输入。

分析到这里，我们可以想象“如何从这些服务中选择一个服务进行调用？”这个问题应该交给 IRule 去解决，因为负载均衡算法决定了选择服务的结果。那么针对第一个问题“服务器列表中有哪些服务？”，我们就需要联想到服务注册中心 Eureka，因为 Eureka 中存放着系统中当前所有的服务注册信息。

Ribbon 中动态加载后端服务列表是通过 DynamicServerListLoadBalancer 类实现的，它

仅供非商业用途或交流学习使用

是前面提到的 BaseLoadBalancer 的子类。在 DynamicServerListLoadBalancer 类中，首先依靠 EurekaClient 从服务注册中心获取具体的服务实例 InstanceInfo 列表，然后对这个列表进行遍历，将状态为在线的实例转换成 DiscoveryEnabledServer 对象并放到一个集合中，最后将这个集合返回。DiscoveryEnabledServer 对象的集合就是 Ribbon 进行负载均衡所需的服务器列表。通过 DynamicServerListLoadBalancer 获取服务列表的时序图如图 4-8 所示。

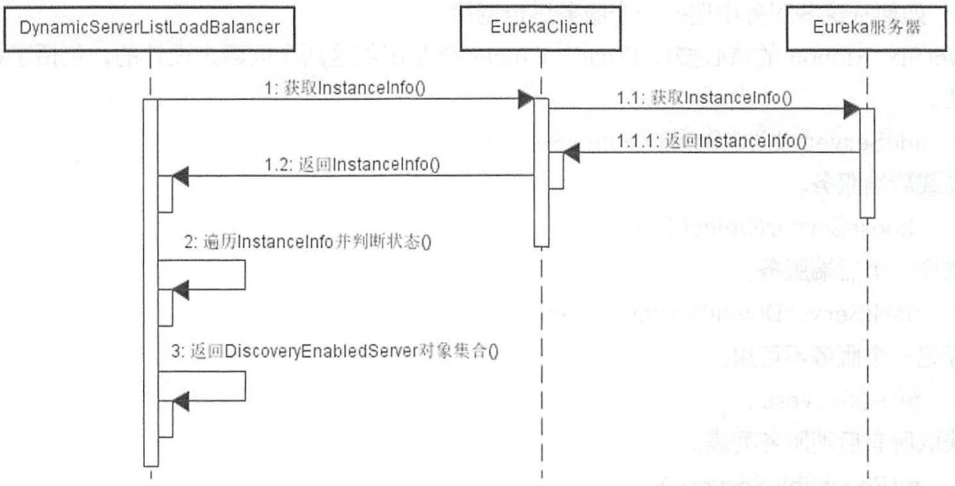


图 4-8　DynamicServerListLoadBalancer 获取服务列表时序图

从图 4-8 中我们看到整个过程与 4.2.2 节中使用 DiscoveryClient 查找服务的流程类似，实际上 EurekaClient 的实现类就是 DiscoveryClient，DiscoveryClient 具有服务注册、获取服务注册列表等核心功能。由此可见，DynamicServerListLoadBalancer 是从 EurekaClient 获取服务注册信息并根据 IRule 去路由，然后根据 IPing 去判断服务的可用性，最终返回一个当前可用的服务列表。

在 3.1.2 节讲解服务治理的基本方案时提到客户端从 Eureka 服务器获取服务列表的方式有两种，一种是轮询机制，另一种是监听和通知机制。根据本节的分析，显然 DynamicServerListLoadBalancer 用的是前者。既然使用的是轮询机制，就需要明确 DynamicServerListLoadBalancer 采用何种频率通过 EurekaClient 获取服务注册信息。实际上，这里采用的策略是通过 10 秒一次的频率向 EurekaClient 发送 Ping 请求来判断服务的可用性，如果服务的可用性发生了变化或者服务数量和之前的不一致，则更新或者重新获取服务注册信息，反之则不会主动去获取服务列表。

Ribbon 还有一个功能是区域亲和性，该功能与 Eureka 所提供的 Region 和 Zone 划分机制相关，关于 Region 和 Zone 的讨论请参考 3.4.3 节中的内容。当 Ribbon 动态从 Eureka 获取注册服务集合后，还会用一个过滤器对得到的服务器列表再次进行过滤，根据配置会选择和服务消费者在一个 Zone 的服务，显然这样可以降低服务调用的延迟。在 DynamicServerListLoad

Balancer 中，这个过滤器就是 ZoneAffinityServerListFilter 类。

判断是否开启区域感知功能也有一点需要注意，如果过滤后的服务实例当前运行状态比较忙，则停止区域感知。也就是说 Ribbon 实现区域亲和性是基于实时的服务器运行状况以决定是否只选择本区域的服务。至于如何获取服务器运行状况，显然就需要使用到负载均衡器自身所具备的 LoadBalancerStats 类，我们在前面已经提到过这个工具类。

4.3.2 Ribbon 负载均衡策略

当获取服务列表之后，我们就可以采用一定的负载均衡策略解决“如何从这些服务中选择一个服务进行调用？”这一问题。Ribbon 把这个工作委托给了 IRule 接口的实现类，不同的 IRule 实现类可以提供不同的负载均衡算法。

IRule 接口包含 3 个方法，其中最重要的 choose()方法根据 key 来获取后台服务，而 setLoadBalancer()和 getLoadBalancer()方法则用来设置和获取 ILoadBalancer。通过实现 IRule 接口，Ribbon 提供了常用的几种负载均衡策略，包括 RoundRobinRule、RandomRule 这样不依赖于服务器运行状况的静态策略，也有 AvailabilityFilteringRule、WeightedResponseTimeRule 等多种基于收集到的服务器运行状况进行决策的动态策略。Ribbon 中常见的负载均衡策略如下所示。

- RoundRobinRule 策略

以轮询的方式依次请求不同的服务器，即每次调度执行 $i=(i+1)\bmod n$ 操作从而选择第 i 台服务器，这里的 mod 为取模操作。

- RandomRule 策略

随机选择状态为可用的服务。

- WeightedResponseTimeRule 策略

根据响应时间分配一个权重，响应时间越长，权重越小，被选中的可能性就越低。一个后台线程定期从 LoadBalancerStats 读取平均响应时间，为每个服务器计算一个权重。权重的计算也比较简单，即每次请求的响应时间减去每个服务平均的响应时间就是该服务的权重。

- RetryRule 策略

对选定的负载均衡策略执行重试机制，在一个可配置的时间段内，每当选择服务器不成功，就一直尝试选择一个可用的服务器。

- AvailabilityFilteringRule 策略

通过检查 LoadBalancerStats 中记录的各个服务器的运行状态，过滤掉那些处于一直连接失败或处于高并发状态的后端服务器。

- BestAvailableRule 策略

选择一个并发请求量最小的服务器，逐个考察服务器后选择其中活跃请求数最小的服务器。

- ZoneAwareRule 策略

区域感知负载均衡内置电路跳闸逻辑，可被配置基于区域亲和性来选择目标服务实例。它监控每个区域中服务实例的行为，当本区域发生故障时可以快速完成切换。

4.3.3 @LoadBalanced 注解与 RestTemplate

到现在为止，可能还有一个问题没有得到解答，即为什么通过@LoadBalanced 注解创建的 RestTemplate 就能自动具备客户端负载均衡的能力。我们知道 RestTemplate 是 Spring 自带的一个支持 RESTful 风格的 API 调用工具类，而@LoadBalanced 注解就是用来给 RestTemplate 添加一种修饰，以便通过拦截的方式将代码执行流程导向负载均衡客户端 LoadBalanceClient 类。

事实上，在 Ribbon 中存在一个 LoadBalancerAutoConfiguration 类，即 LoadBalancer 的自动配置类。在该类中，首先维护了一个被@LoadBalanced 修饰的 RestTemplate 对象的列表，在初始化的过程中，对目标 RestTemplate 增加拦截器 LoadBalancerInterceptor。LoadBalancerInterceptor 用于实时拦截，而在它的拦截方法本质上是使用 LoadBalanceClient 来执行真正的负载均衡。

在 Ribbon 中，LoadBalancerClient 是一个非常重要的接口，它继承自 ServiceInstance Chooser 接口并被 RibbonLoadBalancerClient 类所实现，这三者之间的关系如图 4-9 所示。

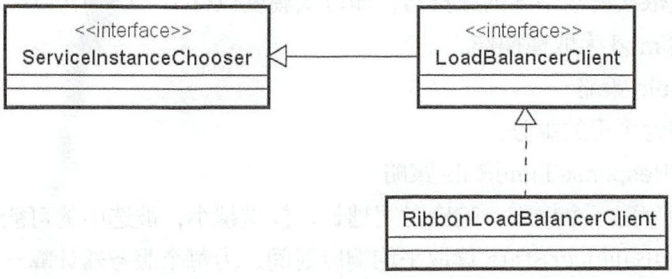


图 4-9 LoadBalancerClient 相关类层结构图

LoadBalanceClient 的定义也比较简单，只有如下 3 种方法。

- ServiceInstance choose(String serviceId)

选择服务实例，根据传入的服务名 serviceId 从负载均衡器中挑选一个对应服务的实例。

- T execute(String serviceId, LoadBalancerRequest request)

执行服务调用，使用从负载均衡器中挑选出的服务实例来执行请求内容。

- URI reconstructURI(ServiceInstance instance, URI original)

构建服务 URI，为系统构建一个合适的 host:port 形式的 URI。使用负载均衡所选择的

ServiceInstance信息重新构造访问URI，也就是用服务实例的host和port再加上服务的端点路径来构造一个真正可供访问的服务。

图4-9中的ServiceInstanceChooser接口的方法只有一个，即ServiceInstance choose(String serviceId)，该方法可根据serviceId来获取ServiceInstance。

RibbonLoadBalancerClient类负责对请求进行最终的负载均衡处理。在RibbonLoadBalancerClient中，choose()方法负责选择具体服务实例。当发起一个请求时，负载均衡根据输入的serviceId选择一个服务实例，最终调用的也是Netflix Ribbon中ILoadBalancer接口所定义的chooseServer()方法。

如下代码演示了直接使用LoadBalancerClient进行服务访问的过程，通过注入LoadBalancerClient并使用它的choose()方法，同样可以对Product服务发起调用并获取ServiceInstance。

```
@Autowired
private LoadBalancerClient loadBalancerClient;

public void userLoadBalancerClient() {
     ServiceInstance serviceInstance
         = loadBalancerClient.choose("productservice");

     serviceInstance.getServiceId();
     serviceInstance.getHost();
     serviceInstance.getPort();
     serviceInstance.getUri();
     ...
}
```

至此，从@LoadBalanced注解到Ribbon的内部核心组件再到Eureka注册中心的完整流程已经介绍完毕，整个运行机制和工作流程也为我们实现客户端负载均衡提供了一套设计思路和解决方案。

4.3.4 @RibbonClient注解与自定义负载均衡策略

在本章的最后，还将通过@RibbonClient注解来实现更加精细化的负载均衡策略，该注解的使用需要对Ribbon基本架构和原理有一定了解。

Spring CloudNetflix Ribbon提供@RibbonClient注解的目的在于通过该注解声明自定义配置来完全控制客户端的负载均衡行为。

以下代码演示的就是一个自定义的配置类RibbonConfig，显然该配置类的作用是使用RandomRule替换Ribbon中的默认负载均衡策略，可以根据需要返回任何自定义的IRule接口

的实现策略。

```
@Configuration
public class RibbonConfig {

    @Autowired
    IClientConfig config;

    @Bean
    @ConditionalOnMissingBean
    public IRule ribbonRule(IClientConfig config) {

        return new RandomRule();
    }
}
```

有了 RibbonConfig 之后，我们就可以在调用特定服务时使用该配置类，从而对客户端负载均衡实现细粒度的控制。在 Order 服务中使用 RibbonConfig 实现对 Product 服务访问的示例代码如下所示，现在每次访问 Product 服务都会使用 RandomRule 负载均衡策略。

```
@SpringBootApplication
@EnableEurekaClient
@RibbonClient(name = "productservice", configuration =
    RibbonConfig.class)
public class OrderApplication {

    @Bean
    @LoadBalanced
    public RestTemplate restTemplate(){
        return new RestTemplate();
    }

    public static void main(String[] args) {
        SpringApplication.run(OrderApplication.class, args);
    }
}
```

对比@LoadBalanced 注解和@RibbonClient 注解，如果使用的是服务注册和发现机制，那么通常只需要@LoadBalanced 注解就能完成客户端负载均衡。如果我们没有使用类似 Eureka 的服务注册中心或者需要对 Ribbon 运行行为进行定制时，那么就需要用到@RibbonClient 注解。

4.4 本章小结

本章介绍了 Spring Cloud 中用于提供负载均衡功能的 Spring Cloud Netflix Ribbon 组件。在 Spring Cloud 中，Ribbon 组件与第 3 章介绍的 Eureka 组件关系密切，需要相互结合作为一个整体进行理解和掌握。

Ribbon 是一种客户端负载均衡工具，本章第一部分对负载均衡的类型和基本算法做了简要展开，帮助读者了解工具背后的背景和设计思想。

本章第二部分介绍了 Ribbon 的使用方法和工作流程，可以看到 Spring Cloud 已经为我们提供了简单的集成方式，通过对现有 RestTemplate 工具类进行改造，我们不需要做额外的开发工作就能在代码中自动嵌入客户端负载均衡机制。

工具的使用固然简单，但还是需要对工具内部的实现机制做深入了解。本章最后对 Ribbon 的基本架构展开讨论，分析了 Ribbon 中几个核心接口的定义以及关键注解的工作原理。

第 5 章

Spring Cloud Netflix Hystrix 与服务容错

在微服务架构中，各个服务独立部署且服务与服务之前存在相互依赖关系。与单块系统相比，微服务架构中出现服务访问失败的原因和场景非常复杂，需要我们从服务可靠性的角度出发对服务自身以及服务与服务之间的交互过程进行设计。服务可靠性是微服务架构的关键要素之一，本章从容错（Fault Tolerance）思想入手，分析 Spring Cloud 框架中确保服务可靠性的各种技术和方法。

服务可靠性问题同时涉及服务的提供者和消费者。对于服务提供者而言，要做的事情比较简单，一旦自身服务发生错误，那么应该快速返回合理的处理结果，也就是要做到快速失败（Failfast）。而对于服务消费者而言，事情就没有那么简单了，一方面可以采用超时（Timeout）和重试（Retry）等常见方法，另一方面，业界也有一些模式应对服务提供者发生失败的场景。

在学习了第 3 章介绍的注册中心及第 4 章介绍的客户端负载均衡之后，本章关注于服务消费者容错机制。我们将在各个服务调用的过程中添加消费者容错机制，这些机制包括服务隔离（Isolation）、服务熔断（Circuit Breaker）和服务回退（Fallback）。图 5-1 展示了系统架构演进的这一过程。

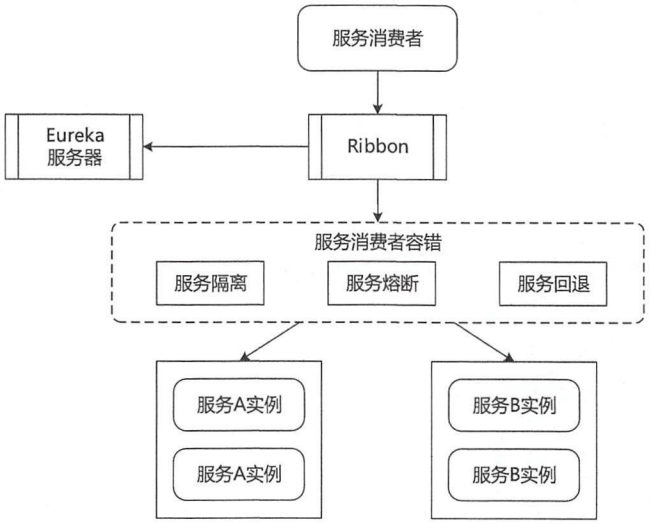


图 5-1　添加服务消费者容错之后的系统架构图

针对消费者容错的需求和基本实现模式，本章将引入 Spring Cloud 中非常强大的一个组件 Spring Cloud Netflix Hystrix。Spring Cloud Netflix Hystrix 基于 Netflix Hystrix 并做了封装，在后面的示例中，我们会发现使用它来实现服务隔离、服务熔断和服务回退的过程是如此简单和便捷。

5.1 服务消费者容错思想和模式

本节首先分析需要确保服务消费者实现容错的原因，然后引出各种实现服务容错的模式。理解这些概念和模式有助于更好地使用 Spring Cloud Netflix Hystrix。

5.1.1 服务消费者容错的需求

在微服务架构中，对于服务调用的双方而言，服务提供者可能会发生失败，由于设计实现上考虑不周、代码中存在缺陷等因素所造成的服务失败场景也不少见，这些场景不是本书讨论的重点。除了服务自身失败，我们需要考虑的另一种失败表现形式是服务依赖失败。服务依赖失败较之服务自身失败而言影响更大，也更加难以发现和处理。服务依赖失败是我们在设计微服务架构时需要重点考虑的服务可靠性因素，因为服务依赖失败会造成失败扩散，从而形成服务访问的雪崩效应（Avalanche Effect）。雪崩效应是我们引入服务消费者容错思想和模式的根本需求，在讨论具体的应对策略之前，我们首先需要明确其产生的原因。

图 5-2（边框为虚线的服务代表不可用）展示了雪崩效应的效果图，可以看到 A、B、C、

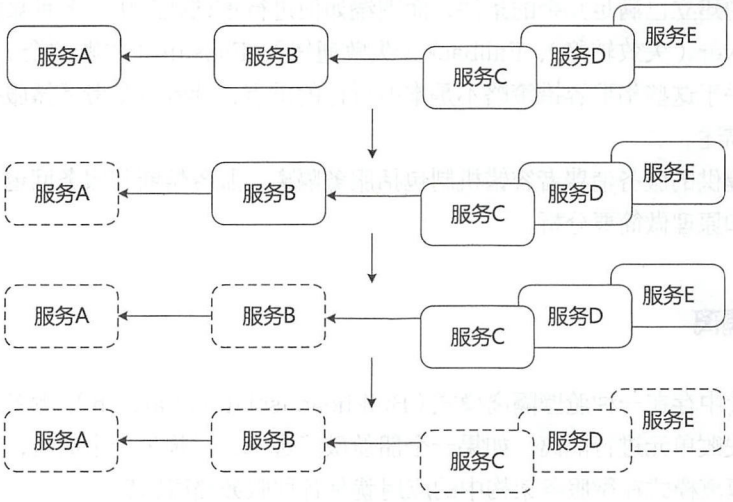


图 5-2 服务雪崩效应效果图

D、E 共 5 个服务存在依赖关系，A 为服务提供者，B 为 A 的服务消费者，C、D 和 E 是 B 的服务消费者。假如 A 变成不可用，就会引起 B 的不可用，并将这种不可用性逐渐扩散到 C、D 和 E，从而造成了整个服务体系发生雪崩。

服务雪崩的产生是一种扩散效应，我们可以对图 5-2 中的现象进行剥离，先从 A 和 B 两个服务之间的交互展开讨论。图 5-3（边框为虚线的服务代表不可用）展示了雪崩效应产生的 3 个阶段，即首先服务提供者 A 发生不可用，然后服务消费者 B 不断进行重试加大了访问流量，最后导致 B 服务自身也不可用。

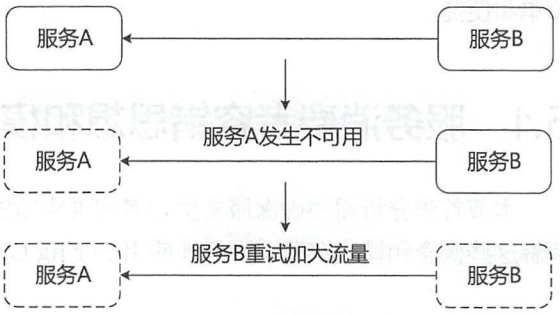


图 5-3 两个服务交互图

在图 5-3 中，B 服务通过用户不断提交服务请求等手工重试或代码逻辑自动重试等手段进一步加大对 A 服务的访问流量。因为 B 服务使用同步调用，会产生大量的等待线程占用系统资源。一旦线程资源被耗尽，B 服务提供的服务本身也将处于不可用状态。这一效果在整个服务访问链路上进行扩散，就形成了雪崩效应。

显然，应对雪崩效应的切入点不在于服务提供者，而在于服务消费者。我们不能保证所有服务提供者都不会失败，但是我们要想办法确保服务消费者不受已失败的服务提供者的影响，或者说需要将服务消费者所受到的这种影响降到最低，这就是服务消费者容错的需求。

在第 3 章中介绍了集群和客户端负载均衡，从消费者容错的角度讲，负载均衡不失为一种好的容错策略。容错机制的基本思想是冗余和重试，即当一个服务器出现问题时不妨试试其他服务器。集群的建立已满足冗余的条件，而围绕如何进行重试就产生了各种集群容错策略，常见的包括 Failover（失效转移）、Failback（失败通知）、Failsafe（失败安全）和 Failfast（快速失败）等。关于这些集群容错策略不是本书讨论的重点，读者可参考《微服务设计原理和架构》中的相关描述。

本章所要提供的服务消费者容错机制包括服务隔离、服务熔断和服务回退，下面对这些机制的基本概念和原理做简要分析。

5.1.2 服务隔离

在架构设计中存在一种舱壁隔离模式（Bulkhead Isolation Pattern），顾名思义就是像舱壁一样对资源或失败单元进行隔离，如果一个船舱破了进水，只损失一个船舱，其他船舱可以不受影响。舱壁隔离模式在微服务架构中的应用就是各种服务隔离思想。

服务隔离包括一些常见的隔离思路以及特定的隔离实现技术框架。所谓隔离，本质上是对

系统或资源进行分割，从而实现当系统发生故障时能够限定传播范围和影响范围，即发生故障后只有出问题的服务不可用，而保证其他服务仍然可用。隔离的基本思路如图 5-4 所示。

1. 线程隔离

线程隔离主要通过线程池（Thread Pool）进行隔离，在实际使用时我们会把业务进行分类并交给不同的线程池进行处理。当某个线程池处理一种业务请求发生问题时，不会将故障扩散到其他线程池，也就不会影响到其他线程池中所运行的业务，从而保证其他服务可用。在图 5-4 中，把隔离媒介替换成线程池就能起到线程隔离的效果。

线程隔离是实现服务隔离的常见手段，下面通过一个实例来进一步介绍它的工作场景。假如系统存在商品服务、用户服务和订单服务 3 个微服务，然后通过设置运行时环境参数得到这 3 个服务一共使用 200 个线程，客户端调用这 3 个服务会共享线程池，如图 5-5 所示。

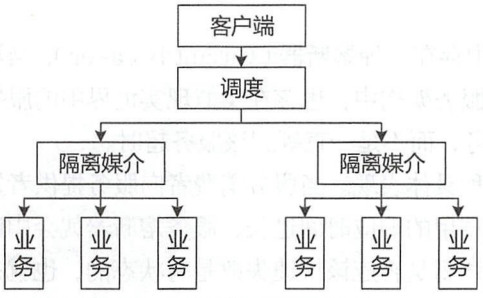


图 5-4　隔离的基本思路

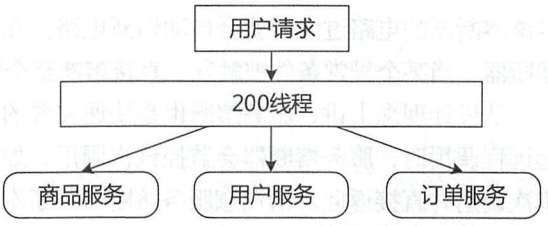


图 5-5　没有使用线程池隔离的场景

在图 5-5 中，如果其中的商品服务不可用，就会出现线程池里所有线程都因同步等待响应而被阻塞，从而造成服务雪崩，如图 5-6 所示（边框为虚线的服务代表不可用），可以看到因为商品服务不可用导致共享的 200 线程全部耗尽，从而影响了用户服务和订单服务。

线程隔离机制通过为每个依赖服务分配独立的线程池以实现资源隔离（见图 5-7）。在图 5-7 中，当商品服务不可用时，即使为商品服务独立分配的 50 个线程全部处于同步等待状态，也不会影响其他服务的调用，因为其他服务运行在独立的线程池中。

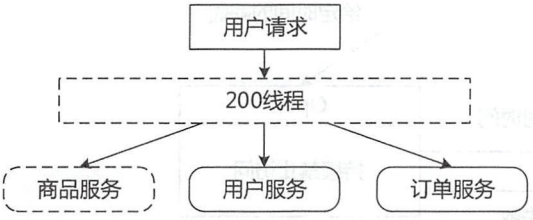


图 5-6　没有使用线程池隔离造成的服务雪崩场景

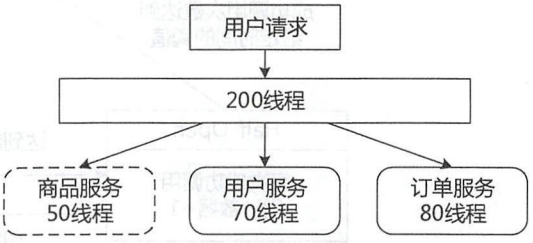


图 5-7　使用线程池隔离的场景

2. 进程隔离

进程隔离比较好理解，就是将系统拆分为多个子系统以实现物理隔离，各个子系统运行在独立的容器和 JVM 中，通过进程隔离使得某一个子系统出现问题不会影响到其他子系统。图 5-4 中，把隔离媒介替换成 JVM 就能起到进程隔离的效果。从进程隔离的角度讲，对系统进行微服务建模和拆分就是一种具体的实现方式，每个服务独立部署和运行，各个服务之间实现了物理隔离。服务隔离还包括集群隔离、机房隔离和读写隔离等其他表现形式。

Spring Cloud Netflix Hystrix 实现了线程隔离机制，同时还提供了基于信号量的隔离方案，本章后续内容会对这两个隔离方式进行详细展开。

5.1.3 服务熔断

服务熔断的概念来源于电路系统，在电路系统中存在一种熔断器（Circuit Breaker），当流经该熔断器的电路过大时就会自动切断电路。在微服务架构中，也存在类似现实世界中的服务熔断器，当某个异常条件被触发，直接熔断整个服务，而不是一直等到该服务超时。

从设计理念上讲，服务熔断也是快速失败的一种具体表现。当服务消费者向服务提供者发起远程调用时，服务熔断器会监控该次调用，如果调用的响应时间过长，服务熔断器就会中断本次调用并直接返回。请注意服务熔断器判断本次调用是否应该快速失败是有状态的，也就是说服务熔断器会把所有的调用结果都记录下来，如果发生异常的调用次数达到一定的阈值，那么服务熔断机制才会被触发，快速失败就会生效；反之将按照正常的流程执行远程调用。

对以上过程进行抽象和提炼，可以得到如图 5-8 所示的服务熔断器基本结构，该结构给出了熔断器实现上的 3 个状态机。

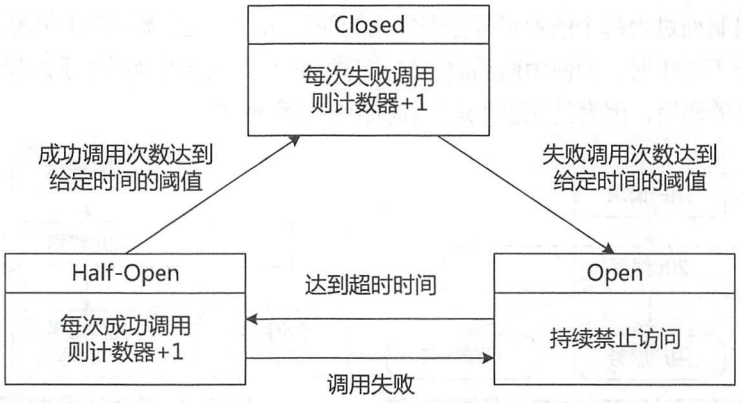


图 5-8 服务熔断器结构图

- Closed

熔断器关闭状态，不对服务调用进行限制，但会对调用失败次数进行积累，到达一定阈值或比例时会启动熔断机制。

- Open

熔断器打开状态，此时对服务的调用将直接返回错误，不执行真正的网络调用。同时，熔断器设计了一个时钟选项，当时钟达到了一定时间（这个时间一般设置成平均故障处理时间，也就是MTTR）时会进入半熔断状态。

- Half-Open

半熔断状态，允许一定量的服务请求，如果调用都成功或达到一定比例则认为调用链路已恢复，关闭熔断器；否则认为调用链路仍然存在问题，又回到熔断器打开状态。

Spring Cloud Netflix Hystrix 同样实现了一个服务熔断器，具备与图 5-8 类似的结构和功能。

5.1.4 服务回退

服务回退（Fallback）在处理因为服务依赖而导致的异常时也是一种有效的容错机制。当远程调用发生异常时，服务回退不是直接抛出该异常，而是产生另外的处理机制来应对该异常，相当于执行了另一条路径上的代码或返回一个默认处理结果，而这条路径上的代码或这个默认处理结果一定满足业务逻辑的实现需求，而只是告知服务的消费者当前调用中所存在的问题。显然，服务回退不能解决由异常引起的实际问题，它只是一种权宜之计。

在现实环境中，服务回退的实现方式可以很简单，原则上只需要保证异常被捕获并返回一个处理结果即可。但在有些场景下，回退的策略则非常复杂，我们可能会从其他服务或数据中获取相应的处理结果，需要具体问题具体分析。

Spring Cloud Netflix Hystrix 支持服务回退，通过自定义回退方法（Fallback Method）这一特性提供了一种非常简单的机制来支持服务回退。

5.2 使用 Hystrix 实现服务容错

Spring Cloud Netflix Hystrix 基于 Netflix Hystrix 实现了服务隔离、服务熔断和服务回退 3 种服务容错机制。本节首先引入 Hystrix 框架，然后分别提供如何应用 Hystrix 实现 3 种服务容错机制的方法和示例。

5.2.1 引入 Hystrix

Netflix Hystrix 是 Netflix 开源的一款针对分布式系统的延迟和容错库，可用来隔离分布式服务故障，以确保系统的可用性。

在 Netflix Hystrix 中，HystrixCommand 是重中之重，在 Netflix Hystrix 的整个机制中涉及依赖边界的地方，都是通过这个 HystrixCommand 进行封装。我们在使用 HystrixCommand 时需要提供它的子类，子类要实现的方法主要有以下两个。

- run()方法

该方法用于实现所依赖的业务逻辑，或者说是实现微服务之间的调用。

- getFallBack()方法

该方法用于实现服务回退处理逻辑。

Spring Cloud 内部集成了 Netflix Hystrix 框架，与其他 Spring Cloud 中的组件类似，Spring Cloud Netflix Hystrix 也提供了专门的注解构建 Hystrix 客户端。实现过程中，使用 Hystrix 命令模式封装依赖逻辑并提供 Fallback 方法实现服务回退策略。为了简化描述，下文中使用 Hsytrix 代表 Spring Cloud Netflix Hystrix。

要使用 Hystrix，首先需要在服务消费者端添加相应的 maven 依赖。在本书的案例中，我们知道 Order 服务会调用 Product 服务和 Account 服务，所以 Order 服务就是需要添加容错机制的服务消费者，我们在 Order 服务的 pom 文件中添加如下内容。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
```

基于上一章的结果，在 OrderApplication 的 Bootstrap 类中添加一个新的注解 @EnableCircuitBreaker，完整的 OrderApplication 类如下所示。

```
@SpringBootApplication
@EnableEurekaClient
@EnableCircuitBreaker
public class OrderApplication {

    @LoadBalanced
    @Bean
    public RestTemplate getRestTemplate(){
        return new RestTemplate();
    }
```

```
    public static void main(String[] args) {
        SpringApplication.run(OrderApplication.class, args);
    }
}
```

@EnableCircuitBreaker注解的作用在于告诉Spring Cloud希望在该服务中启用Hystrix。添加该注解的效果就相当于在Order服务中自动注入了服务熔断器，并可以使用服务隔离和服务回退功能。

5.2.2 使用Hystrix实现服务隔离

针对服务隔离，Hystrix组件提供了两种解决方案，即线程池（Thread Pool）隔离和信号量（Semaphore）隔离，这两种隔离方式都是限制对共享资源的并发访问量。

1. Hystrix线程池隔离

Hystrix线程池运行时效果如图5-9所示，可以看到它为不同的服务提供了独立的线程池，这与5.1.2节中介绍的线程隔离机制保持一致。

在使用方式上，可以引入Hystrix依赖，并使用命令模式封装业务逻辑。例如，使用线程池构建HystrixCommand的代码风格如下所示。

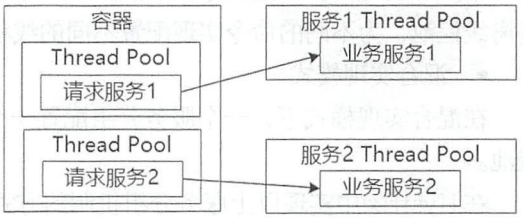


图5-9 Hystrix线程池示意图

```
public class GetOrderCommand extends HystrixCommand<List> {

    private OrderService orderService;

    protected GetOrderCommand(String name) {
        super(Setter.withGroupKey(
            HystrixCommandGroupKey.Factory.asKey("OrderGroup"))
        .andCommandKey(HystrixCommandKey.Factory.asKey("orderKey"))
        .andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey(name)))
            .andCommandPropertiesDefaults(
                HystrixCommandProperties.Setter()
                    .withExecutionTimeoutInMilliseconds(5000))
            .andThreadPoolPropertiesDefaults(
                HystrixThreadPoolProperties.Setter()
                    .withMaxQueueSize(10)
                    .withCoreSize(2)
```

```
                )
            );
        }

        @Override
        protected List run() throws Exception {
            return orderService.getOrderList();
        }
    }
```

当采用线程池隔离时，Hystrix 可以通过粗粒度实现隔离，也可以通过细粒度实现隔离，包括服务分组+线程池、服务分组+服务+线程池和混合实现等 3 种模式。

- 服务分组+线程池模式

服务分组+线程池模式是一种粗粒度实现，一个服务分组配置一个隔离线程池即可。可以不配置线程池名称，也可以将相同分组的线程池名称配置得一样。

- 服务分组+服务+线程池模式

服务分组+服务+线程池模式则是一种细粒度实现，一个服务分组中的每一个服务配置一个隔离线程池，为不同的命令实现配置不同的线程池名称即可。

- 混合实现模式

在混合实现模式下，一个服务分组配置一个隔离线程池，然后对重要服务单独设置隔离线程池。

在 Hystrix 中实现以上服务分组机制的方式是在初始化 HystrixCommand 时设置相应的键值。Hystrix 提供了用于配置全局唯一标识服务分组名称的 HystrixCommandGroupKey，相同分组的服务会聚合在一起；HystrixCommandKey 用于配置全局唯一标识服务的名称，可以为每个服务起一个全局唯一的名字；HystrixThreadPoolKey 则用于配置全局唯一标识线程池的名称，相同的线程池名称代表使用的是同一个线程池。

同时，Hystrix 还可以通过 HystrixThreadPoolProperties 配置线程池参数，例如，使用 coreSize 配置核心线程池的最大值，使用 maxQueueSize 配置线程池队列的最大值，使用 keepAliveTimeMinutes 指定线程池中空闲线程生存时间等。

回到案例，在 Order 服务的 OrderService 类中，我们会找到如下所示的 getOrders()方法，该方法调用 orderRepository 从数据库中查询分页数据。可以看到这里使用了 @HystrixCommand 注解来完成 HystrixCommand 的创建，这是使用 HystrixCommand 的另一种方法。该 HystrixCommand 设置了 threadPoolKey，也提供了 threadPoolProperties 来设置 coreSize 和 maxQueueSize。

```
@HystrixCommand(threadPoolKey = "orderThreadPool",
    threadPoolProperties =
          {@HystrixProperty(name = "coreSize",value="30"),
```

仅供非商业用途或交流学习使用

```
            @HystrixProperty(name="maxQueueSize", value="10")}
)
public List<Order> getOrders(int pageIndex, int pageSize) {
    return orderRepository.findAll(
        new PageRequest(pageIndex - 1, pageSize)).getContent();
}
```

线程池的主要缺点在于它增加了计算的开销，每个被包装成命令的业务请求在执行时都会涉及请求排队、调度和上下文切换等操作。对于不依赖网络访问的服务，比如在只依赖于内存或缓存的情况下，不适合使用线程池隔离技术，推荐使用信号量隔离。

2. Hystrix 信号量隔离

线程池隔离和信号量隔离的主要区别在于线程池方式下业务请求线程和执行依赖服务的线程不是同一个线程，而信号量方式下这两者使用的是同一个线程。

Hystrix 同样提供了 HystrixCommandProperties 用于配置命令的一些参数，如果将属性 execution.isolation.strategy 设置为 SEMAPHORE，则 Hystrix 使用信号量而不是默认的线程池来做隔离。使用信号量构建 HystrixCommand 的代码风格如下。

```
public class GetOrderCommand extends HystrixCommand<List>{

    private OrderService orderService;

    public GetOrderCommand(String name) {
        super(Setter.withGroupKey(

    HystrixCommandGroupKey.Factory.asKey("OrderGroup"))
        .andCommandPropertiesDefaults(
            HystrixCommandProperties.Setter()
            .withExecutionIsolationStrategy(
                ExecutionIsolationStrategy.SEMAPHORE)));
        );
    }

    @Override
    protected List run() throws Exception {
        return orderService.getOrderList();
    }
}
```

在上面的代码中，我们也看到了 HystrixCommandGroupKey，说明信号量模式下也可以实现服务的分组管理。

仅供非商业用途或交流学习使用

5.2.3 使用 Hystrix 实现服务熔断

从 5.1.3 节可以得知熔断器有 3 种状态，其中，打开和半打开状态会导致触发熔断机制。在 Hystrix 中实现熔断器的是 HystrixCircuitBreaker 类，HystrixCommand 首先调用 HystrixCircuitBreaker 中的 allowRequest()方法判断服务是否已经熔断了，而该判断取决于服务访问的失败率。在 Hystrix 中异常、超时、线程池拒绝、信号量拒绝的数量总和就是失败总数，通过失败总数与访问总数之间的对比决定失败率。当该失败率超过一定阈值时就会触发熔断器的打开状态。当熔断处于打开状态后，不能一直熔断下去，需要在一个时间窗口后进行重试，这时候的状态就是半打开。Hystrix 允许在一定的窗口时间内进行一次重试，重试成功则闭合熔断开关，否则熔断开关会一直处于打开状态。

现在来考虑一个服务依赖调用的具体场景。在案例中，我们知道 Order 服务需要调用 Product 服务和 Account 服务来实现下单操作，该操作的代码流程如下所示，可以看到 addOrder()方法会对传入的用户账户 Id 和商品编号进行验证，只有在两者都有效的条件下才能执行订单入库操作。

```
public Order addOrder(Long accountId, String productCode) {
    Order order = new Order();
    order.setId(1L);

    Product product = getProduct(productCode);
    if(product == null) {
        return order;
    }

    Account account = getAccount(accountId);
    if(account == null) {
        return order;
    }

    order.setAccountId(accountId);
    order.setItem(product.getProductName());
    order.setCreateTime(new Date());

    orderRepository.save(order);

    return order;
}
```

显然，上述代码中 getProduct()方法和 getAccount()方法都会涉及微服务之间的相互依赖

仅供非商业用途或交流学习使用

和调用，示例代码如下所示。通过注入 ProductClient 和 AccountClient 两个工具类来实现远程调用，其中，ProductClient 类使用 Ribbon 和 RestTemplate 实现了调用过程的客户端负载均衡，我们在上一章中已经进行了介绍。AccountClient 类的实现方式与 ProductClient 完全相同。

```
@Autowired
ProductClient productClient;

@Autowired
AccountClient accountClient;

private Account getAccount(Long accountId) {
    return accountClient.getAccount(accountId);
}

private Product getProduct(String productCode) {
    return productClient.getProduct(productCode);
}
```

在微服务环境下，使用 ProductClient 和 AccountClient 的调用过程可能会出现响应超时等问题，这个时候 Order 服务作为服务消费者需要做到服务容错。要嵌入 Hystrix 提供的熔断机制，只需要在这两个方法上添加@HystrixCommand 注解即可，如下所示。

```
@HystrixCommand
private Account getAccount(Long accountId) {
    return accountClient.getAccount(accountId);
}

@HystrixCommand
private Product getProduct(String productCode) {
    return productClient.getProduct(productCode);
}
```

现在来模拟一下远程调用超时的场景，调整 getAccount()方法的代码，通过 Thread.sleep(2000)来模拟响应时间过长的场景，如下所示。

```
@HystrixCommand
private Account getAccount(Long accountId) {

    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
```

仅供非商业用途或交流学习使用

```
        return accountClient.getAccount(accountId);
    }
```

现在在 Postman 中访问 http://localhost:8083/v1/orders/1/001 端点，可以得到如图 5-10 所示的示意图。同时，在 Order 服务控制台中会看到“java.lang.InterruptedException: sleep interrupted”异常的抛出，而抛出该异常的来源正是 Hystrix。

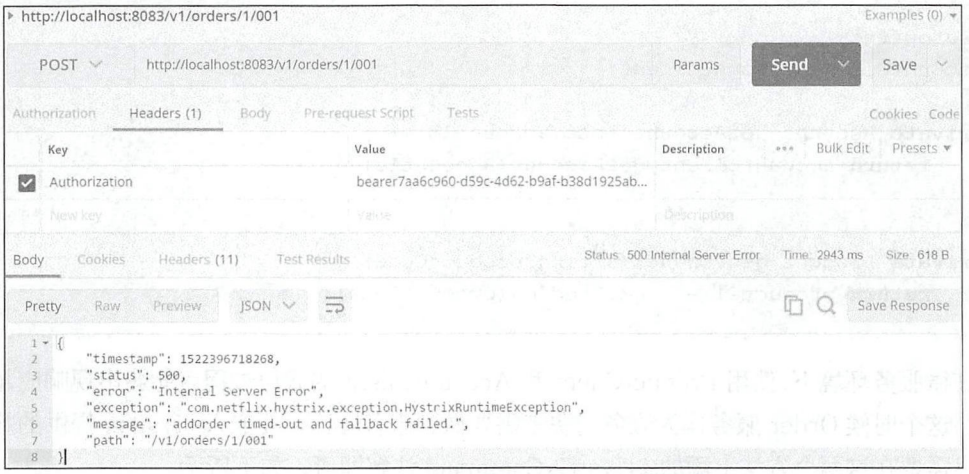


图 5-10　Hystrix 中服务超时抛出异常示意图

在图 5-10 中，我们发现 HTTP 响应状态为 500，而抛出的异常为 HystrixRuntime Exception，从异常信息上可以看出引起该异常的原因是超时。事实上，默认情况下，添加了 @HystrixCommand 注解的方法调用超过了 1000 毫秒就会触发超时异常，显然上例中设置的 2000 毫秒满足触发条件。

和设置线程池及信号量一样，在 HystrixCommand 中也可以对熔断的超时时间、失败率等各项阈值进行设置。例如，可以在 getAccount()方法上添加如下配置项以改变 Hystrix 的默认行为。

```
@HystrixCommand(commandProperties={
    @HystrixProperty(name="execution.isolation.thread.timeoutInMilli
        seconds", value="3000")
    }
)
private Account getAccount(Long accountId) {
    return accountClient.getAccount(accountId);
}
```

上面示例中的 execution.isolation.thread.timeoutInMilliseconds 配置项就是用来设置 Hystrix 的超时时间，现在将其设置成 3000 毫秒，再次访问 http://localhost:8083/v1/

仅供非商业用途或交流学习使用

orders/1/001 端点，在 Postman 中的返回结果如图 5-11 所示，表示调用过程已经成功返回。

图 5-11　使用 Hystrix 正常响应服务请求示意图

Hystrix 提供了多种配置项，关于这些配置项的名称和用法将在 5.3.3 节中进行集中讨论。

5.2.4　使用 Hystrix 实现服务回退

Hystrix 在服务调用失败（异常、拒绝、超时、短路）时可以执行服务回退逻辑。服务回退的整体执行流程如图 5-12 所示，Command 会调用 run()方法，如果 run()方法超时或者抛出异常，则系统会调用 getFallback()方法进行服务回退。

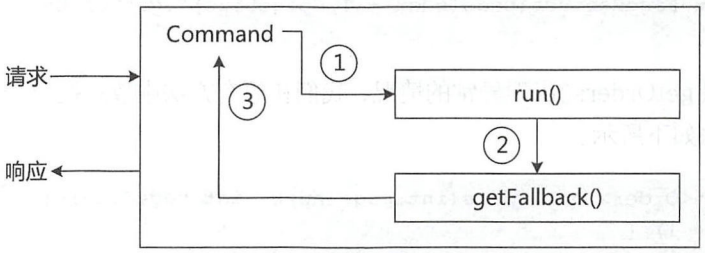


图 5-12　服务回退流程

在开发过程中，我们只需要提供一个 Fallback 方法实现并进行配置即可，对于 getOrder()方法获取订单列表的场景，Fallback 方法示例代码如下所示。请注意，Fallback 方法的参数和返回值必须与真实的 getOrder()方法完全一致。

```
private List<Order> getOrdersFallback(int pageIndex, int pageSize){
    List<Order> fallbackList = new ArrayList<>();
```

仅供非商业用途或交流学习使用

```
        Order order = new Order();
        order.setId(1L);
        order.setAccountId(0L);
        order.setItem("Order list is not available");
        order.setCreateTime(new Date());
        fallbackList.add(order);

        return fallbackList;
    }
```

然后需要在@HystrixCommand 注解中设置“fallbackMethod”配置项，完整版的getOrders()方法代码如下所示，可以看到我们设置了 HystrixCommand 中的线程池、超时参数以及回退方法。

```
    @HystrixCommand(
        fallbackMethod = "getOrdersFallback",
        threadPoolKey = "orderThreadPool",
        threadPoolProperties =
            {@HystrixProperty(name = "coreSize",value="30"),
             @HystrixProperty(name="maxQueueSize", value="10")}
        commandProperties={
            @HystrixProperty(name="execution.isolation.thread.timeoutInM
                illiseconds", value="3000")
        }
    )
    public List<Order> getOrders(int pageIndex, int pageSize) {
        return orderRepository.findAll(
            new PageRequest(pageIndex - 1, pageSize)).getContent();
    }
```

现在来模拟 getOrders()出现异常的情况，我们在原有方法中故意抛出一个异常，修改后的getOrders()方法如下所示。

```
    public List<Order> getOrders(int pageIndex, int pageSize) {
        if(1 == 1) {
            throw new Exception("An exception has occured!");
        }

        return orderRepository.findAll(
            new PageRequest(pageIndex - 1, pageSize)).getContent();
    }
```

上述 getOrders()方法用来模拟服务访问失败的效果，我们在 OrderController 中添加如下入口。

仅供非商业用途或交流学习使用

```
@RequestMapping(value = "/{pageIndex}/{pageSize}", method =
    RequestMethod.GET)
public List<Order> getOrderList( @PathVariable("pageIndex") int
    pageIndex, @PathVariable("pageSize") int pageSize) {

    List<Order> orders = orderService.getOrders(pageIndex, pageSize);
    return orders;
}
```

现在在 Postman 中访问以上入口所返回的结果如图 5-13 所示，我们并没有看到所预期的异常，而是返回了在 getOrdersFallback()方法中定义的结果，说明@HystrixCommand 注解中设置的回退方法已经生效。

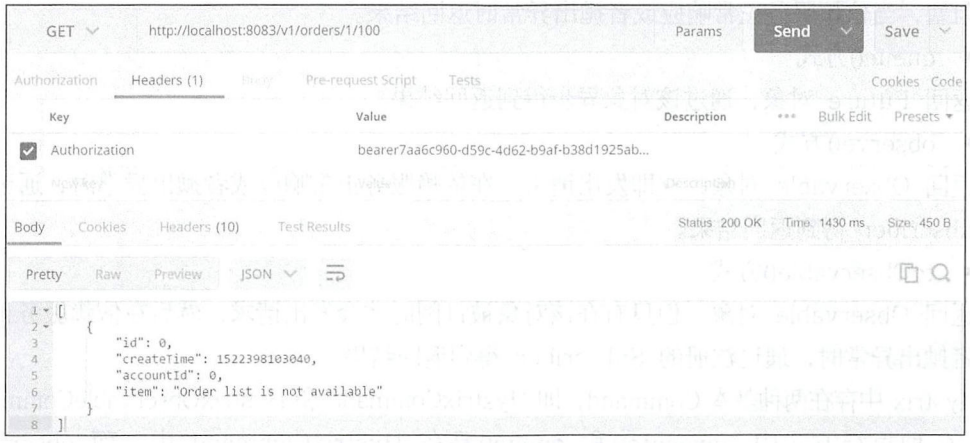


图 5-13　使用 Hystrix 返回 Fallback 结果示意图

5.3　Hystrix 基本原理

介绍完 Hystrix 提供的核心功能之后，本节将从设计和实现的角度出发分析 Hystrix 的基本原理。

5.3.1　服务隔离

Hystrix 使用命令模式的实现类 HystrixCommand 包装依赖调用逻辑，将每个类型的业务请求封装成对应的命令，每个命令在单独线程中执行。创建好的线程池被放入到 ConcurrentHashMap 中，当第二次查询请求过来时，可以直接从 Map 中获取该线程池。具体流程如图 5-14 所示。

仅供非商业用途或交流学习使用

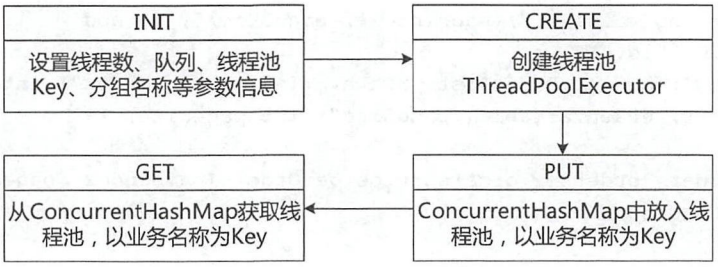


图 5-14　Hystrix 线程池实现流程图

在 Hystrix 中，执行 HystrixCommand 的方式有如下 4 种。

- execute()方式

阻塞，当依赖服务正常响应或者抛出异常时返回结果。

- queue()方式

返回 Future 对象，通过该对象异步得到返回结果。

- observe()方式

返回 Observable 对象，立即发出请求，在依赖服务正常响应或者抛出异常时，通过注册的 Subscriber 得到返回结果。

- toObservable()方式

返回 Observable 对象，但只有在该对象被订阅时才会发出请求，然后在依赖服务正常响应或者抛出异常时，通过注册的 Subscriber 得到返回结果。

Hystrix 中存在两种基本 Command，即 HystrixCommand 和 HystrixObservableCommand。以上 4 种执行方式中，execute()和 queue()是在 HystrixCommand 中，而 observe()和 toObservable()则是在 HystrixObservableCommand 中。从底层实现上讲，HystrixCommand 其实也是利用了 Observable 完成请求的响应。

HystrixCommand 在执行过程中，执行业务代码的线程与请求线程（比如 Tomcat 线程）分离，请求线程可以自由控制离开的时间，这也就是通常所说的异步编程，Hystrix 是结合 RxJava 来实现的异步编程，内部大量使用了 RxJava。RxJava 是一个响应式编程框架，读者可以参考其官方网站做进一步了解。Hystrix 通过设置线程池大小来控制并发访问量，当线程饱和时可以拒绝服务，防止依赖问题扩散（见图 5-15）。

Hystrix 使用线程池存储当前请求以及对请求做出处理。通过设置任务处理超时时间，并将堆积的请求放入线程池队列，可以应对突发流量。当流量洪峰来临时，处理不完的请求可将数据存储到线程池中慢慢处理。

当使用信号量时，Hystrix 使用原子计数器来记录当前运行线程数，新的请求到来时先判断计数器的数值，若超过设置的最大线程数则丢弃该类型的新请求，若不超过则执行计数器+1，请求返回时执行计数器-1。信号量隔离无法应对突发流量。关于线程池和信号量两种隔离机制，

仅供非商业用途或交流学习使用

表5-1做了一个总结。

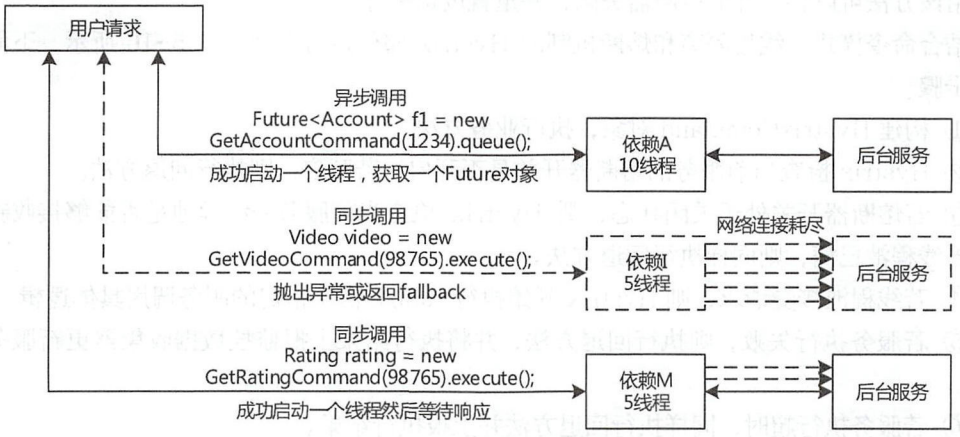


图5-15 Hystrix线程隔离示意图

表5-1 Hystrix两种隔离模式对比

产　品	线程池隔离	信号量隔离
线程	与调用线程隔离	与调用线程相同
成本	排队、调度和上下文切换等开销	无线程切换，成本较低
异步	支持	不支持
并发	基于最大线程池大小	基于最大信号量上限

5.3.2 服务熔断

Hystrix通过HystrixCircuitBreaker接口定义了服务熔断器机制。访问HystrixCircuitBreaker接口有3种方法，分别介绍如下。

- allowRequest()方法

每个请求都通过该方法判断是否可被执行。该方法先根据配置对象properties中的配置信息判断熔断器强制打开或关闭属性是否被设置，如果该属性为强制打开，则直接返回false表示拒绝请求；如果该属性为强制关闭则允许所有请求通过，但同时会调用下面的isOpen()方法来执行熔断器的计算逻辑，用来模拟熔断器打开和关闭的行为。

- isOpen()方法

返回当前熔断器是否打开。如果熔断器打开标识为true则直接返回true。否则将从度量指标对象metrics中获取运行时的统计信息做进一步判断。

- markSuccess()方法

仅供非商业用途或交流学习使用

关闭熔断器。该方法在熔断器处于“半开”状态下使用，如果 Hystrix 命令执行成功，通过调用该方法可以将打开的熔断器关闭，并重置度量指标对象。

结合命令模式、线程隔离和熔断机制的 Hystrix 整体运行流程如图 5-16 所示，下面介绍主要步骤。

① 构建 HystrixCommand 对象，执行业务方法。

② Hystrix 检查当前服务的熔断器开关是否开启，若开启，则执行回退方法。

③ 若熔断器开关处于关闭状态，则 Hystrix 检查当前服务的线程池是否能够接收新的请求，若线程池已满，则同样执行回退方法。

④ 若线程池接受请求，则 Hystrix 开始执行 run()方法中指定的服务调用具体逻辑。

⑤ 若服务执行失败，则执行回退方法，并将执行结果上报监控数据收集器更新服务健康状况。

⑥ 若服务执行超时，同样执行回退方法并上报执行结果。

⑦ 若服务执行成功，返回正常结果。

⑧ 若回退方法执行成功，则返回回退结果。

⑨ 若回退方法执行失败，则抛出异常。

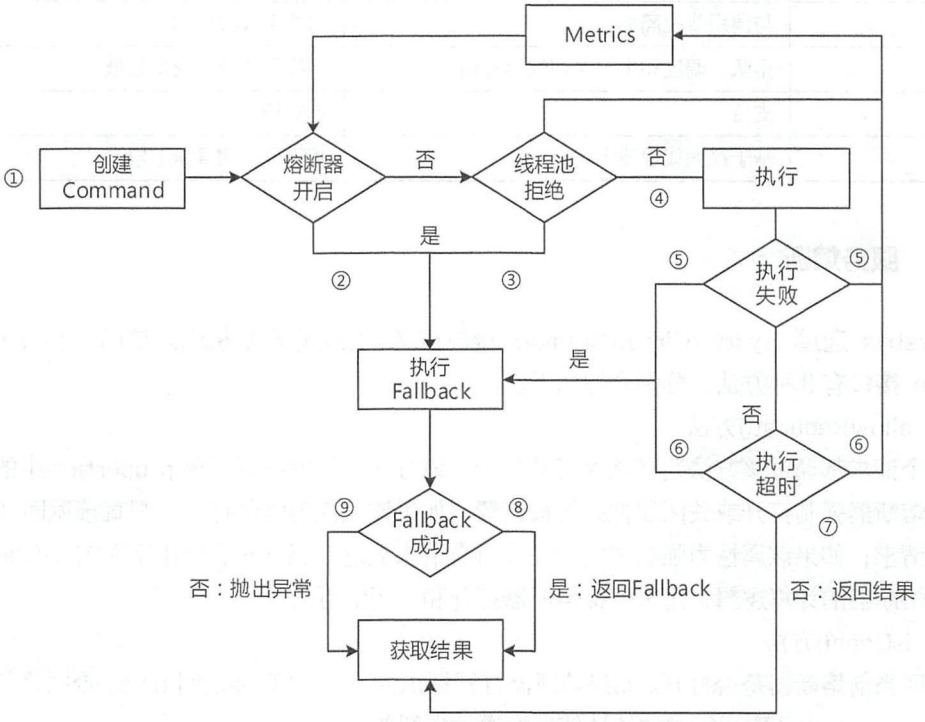


图 5-16 Hystrix 整体运行流程图

仅供非商业用途或交流学习使用

从图 5-16 中可以看到，整个流程涉及熔断器、线程池、监控数据收集器等多个组件，这些组件在运行过程中会执行很多判断条件。作为一个完整而灵活的框架类工具，Hystrix 提供了一批有用的配置项来支持动态改变运行时行为以及实现系统调优。

5.3.3 Hystrix 配置项

有些场景下，Hystrix 的配置项非常有用，例如，我们在 5.2.3 节时演示的服务调用超时，用到的是 execution.isolation.thread.timeoutInMilliseconds 配置项。除此之外，Hystrix 还具备一系列配置项，这些配置项具有层次性，有些配置项适用于整个应用级别，而有些配置项则只适用于某一个类。无论是哪种配置项，一般情况下都有相应的默认值，所以只需要使用 @HystrixCommand 注解就能使这些默认值发挥作用，但我们也很有必要了解这些配置项背后的含义和具体用法。

明确一点，配置项的效果与编写代码的效果是完全一样的。例如，在 5.2.2 节中介绍了使用信号量进行服务隔离的代码风格，也可以采用如下的配置项达到同样的效果。

```
@HystrixCommand(commandProperties={
    @HystrixProperty(name="execution.isolation.strategy",
        value="SEMAPHORE")}
)
```

Hystrix 中存在的主要配置项及其使用方法如下所示，更多关于配置项的介绍读者可参考 Hystrix 的官方网站。

```
hystrix:
  command:
    default:
      execution:
        isolation:
          #线程池隔离还是信号量隔离，默认值为 THREAD，信号量是 SEMAPHORE
          strategy: THREAD
          semaphore:
            #使用信号量隔离时，支持的最大并发数默认值为 10
            maxConcurrentRequests: 10
          thread:
            #command 执行的超时时间，默认值为 1000
            timeoutInMilliseconds: 1000
            #HystrixCommand.run()执行超时时是否被打断，默认为 true
            interruptOnTimeout: true
            #HystrixCommand.run()被取消时是否被打断，默认为 false
            interruptOnCancel: false
        timeout:
```

仅供非商业用途或交流学习使用


```
      #command 执行时间超时是否抛异常，默认为 true
      enabled: true
    fallback:
      #当执行失败或者请求被拒绝，是否会尝试调用 hystrixCommand.getFallback()
      enabled: true
      isolation:
        semaphore:
          #如果并发数达到该设置值，请求会被拒绝和抛出异常并且 fallback 不会被调用，
           默认值为 10
          maxConcurrentRequests: 10

  circuitBreaker:
    #用来跟踪熔断器的健康性，如果未达标则让 request 短路，默认为 true
    enabled: true
    #一个 rolling window 内最小的请求数。如果设为 20，那么当一个 rolling window 的
     时间内（比如说 1 个 rolling window 是 10 秒）收到 19 个请求，即使 19 个请求都失败，
     也不会触发熔断。默认值为 20
    requestVolumeThreshold: 5
    #触发短路的时间值，若该值设为 5000 时，则触发熔断后的 5000 毫秒内都会拒绝 request，
     也就是 5000 毫秒后才会关闭熔断，放部分请求过去。默认值为 5000
    sleepWindowInMilliseconds: 5000
    #错误比率阈值，如果错误率>=该值，熔断会被打开，并且短路所有请求触发 fallback。
     默认值为 50
    errorThresholdPercentage: 50
    #强制打开熔断器，如果打开这个开关就拒绝所有 request，默认为 false
    forceOpen: false
    #强制关闭熔断器，如果这个开关打开，熔断将一直关闭且忽略
    forceClosed: false

  metrics:
    rollingStats:
      #设置统计的时间窗口值的，毫秒值，熔断器的打开会根据 1 个 rolling window 的统计
       来计算。若 rolling window 被设为 10000 毫秒，则 rolling window 会被分成 n 个
       buckets，每个 bucket 包含 success，failure，timeout，rejection 的次数的
       统计信息。默认值为 10000
      timeInMilliseconds: 10000
      #设置一个 rolling window 被划分的数量，若 numBuckets=10，rolling window=
       10000，则一个 bucket 的时间为 1 秒。必须符合 rolling window %numberBuckets==0。
       默认值为 10
      numBuckets: 10
    rollingPercentile:
      #执行时是否 enable 指标的计算和跟踪，默认为 true
      enabled: true
      #设置 rolling percentile window 的时间，默认为 60000
      timeInMilliseconds: 60000
```

仅供非商业用途或交流学习使用


```
        #设置 rolling percentile window 的 numberBuckets。逻辑同上，默认值为 6
        numBuckets: 6
        #如果 bucket size=100，window=10s，若这 10s 里有 500 次执行，只有最后 100 次
         执行会被统计到 bucket 里去。增加该值会增加内存开销以及排序的开销。默认值为 100
        bucketSize: 100
      healthSnapshot:
        #记录 health 快照（用来统计成功和错误率）的间隔，默认值为 500ms
        intervalInMilliseconds: 500
    requestCache:
      #默认 true，需要重载 getCacheKey()，返回 null 时不缓存
      enabled: true
    requestLog:
      #记录日志到 HystrixRequestLog，默认 true
      enabled: true

collapser:
  default:
    #单次批处理的最大请求数，达到该数量触发批处理，默认值为 Integer.MAX_VALUE
    maxRequestsInBatch: 2147483647
    #触发批处理的延迟，也可以为创建批处理的时间+该值，默认值为 10
    timerDelayInMilliseconds: 10
    requestCache:
      #是否对 HystrixCollapser.execute() and HystrixCollapser.queue()的 cache，
       默认 true
      enabled: true

threadpool:
  default:
    #并发执行的最大线程数，默认值为 10
    coreSize: 10
    #Since 1.5.9 能正常运行 command 的最大支付并发数
    maximumSize: 10
    #BlockingQueue 的最大队列数，设为-1，会使用 SynchronousQueue，值为正时使用
    LinkedBlcokingQueue。该设置只会在初始化时有效，之后不能修改 threadpool
    的 queue size，除非 reinitialising thread executor。默认值为-1
    maxQueueSize: -1
    #即使 maxQueueSize 没有达到，达到 queueSizeRejectionThreshold 该值后，请求也会被
    拒绝。因为 maxQueueSize 不能被动态修改，这个参数将允许用户动态设置该值。
    if maxQueueSize == -1，该字段将不起作用
    queueSizeRejectionThreshold: 5
    #Since 1.5.9 该属性使 maximumSize 生效，值须大于等于 coreSize，当设置 coreSize
    小于 maximumSize
    allowMaximumSizeToDivergeFromCoreSize: false
    #如果 corePoolSize 和 maxPoolSize 设成一样（默认实现），则该设置无效
    keepAliveTimeMinutes: 1
```

仅供非商业用途或交流学习使用


```
        metrics:
          rollingStats:
            #线程池统计指标的时间，默认值为 10000
            timeInMilliseconds: 10000
            #将 rolling window 划分为 n 个 buckets，默认值为 10
            numBuckets: 10
```

基于以上配置项，如果我们需要修改 Hystrix 的默认行为，可以采用类似如下的方式对 getAccount()上的@HystrixCommand 注解进行重新设置。

```
@HystrixCommand(commandProperties={
    @HystrixProperty(name="execution.isolation.thread.timeoutInMilli
        seconds", value="12000"),
    @HystrixProperty(name="circuitBreaker.requestVolumeThreshold",
        value="10"),
    @HystrixProperty(name="circuitBreaker.errorThresholdPercentage",
        value="75"),
    @HystrixProperty(name="circuitBreaker.sleepWindowInMilliseconds",
        value="7000")
    }
)
private Account getAccount(Long accountId) {
    return accountClient.getAccount(accountId);
}
```

关于配置项的使用也有两种主流的方法，一种是在@HystrixCommand 注解中加载各种配置项内容，另一种则是将这些配置项从@HystrixCommand 注解中剥离出来放在配置文件中统一管理。从可复用的角度讲，后者更具优势。例如，可以在配置文件中添加如下内容，包括 helloCommand 和 helloGroup 两个自定义的配置项。

```
hystrix:
  command:
    helloCommand:
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: 3000
    default:
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: 5000
  threadpool:
      helloGroup:
```

仅供非商业用途或交流学习使用


```
        coreSize: 1
```

然后在业务代码中，如果需要使用这些配置项内容，则直接引用 helloCommand 和 helloGroup 即可，示例代码如下所示。

```
@HystrixCommand(commandKey = "helloCommand", groupKey = "helloGroup")
public String doSomething() {
    ...
}
```

5.4 本章小结

本章讨论了服务容错机制，并详细介绍了服务消费者容错思想和模式。服务容错是一个很大的主题，也是微服务架构实现过程中必不可少的一个环节。目前市面上关于如何实现服务容错的工具并不是很多，本章介绍的 Spring Cloud Netflix Hystrix 是其中的代表性实现框架。

服务消费者容错的常见实现模式包括服务隔离、服务熔断和服务回退 3 种，Hystrix 对这 3 种模式都有良好的支持。本章先对这 3 种容错模式展开讨论，然后结合案例详细介绍了如何通过 Hystrix 实现服务容错的方法和实践。

本章最后还对 Hystrix 的基本原理做了展开，提供了对服务隔离、服务熔断等模式在实现机制上的分析。同时，也梳理了 Hystrix 中经常会用到的一系列配置项，帮助读者加深对 Hystrix 的理解。

仅供非商业用途或交流学习使用

第 6 章

Spring Cloud Netflix Zuul 与 API 网关

在微服务架构中，API 网关（Gateway）或服务网关（Service Gateway）的出现有其必然性。通常，微服务提供的 API 粒度与客户端的要求不一定完全匹配，微服务一般提供细粒度的 API，这意味着客户端需要与多个服务进行交互才能完成某一个业务功能。更为重要的是，网关能够起到客户端与微服务之间的隔离作用，随着业务需求的变化和时间的演进，网关背后的各个微服务的划分和实现可能需要做相应的调整和升级，这种调整和升级需要实现对客户端透明。

此外，前面章节所介绍的负载均衡机制仍然是有用的，特别是在面向特定服务组合的前置场景中。但是把负载均衡器放置在系统中所有服务的前面往往不是一个很好的解决方案，因为这会导致负载均衡器成为系统运行时的瓶颈，更好的解决方案是在负载均衡器的背后再添加一层网关。

在微服务架构中，我们可以根据需要在服务提供者和消费者之间架设这层 API 网关，从而确保能够满足上文中提到的各种需求。在注册中心和负载均衡的基础上，添加了 API 网关之后的系统架构如图 6-1 所示。当然，并不是所有的服务调用链路都需要添加这层网关，我们也可以根据具体场景直接通过负载均衡器进行服务访问。实际应用过程中，这种混合式的服务调用管理方式也是一种常见的做法。

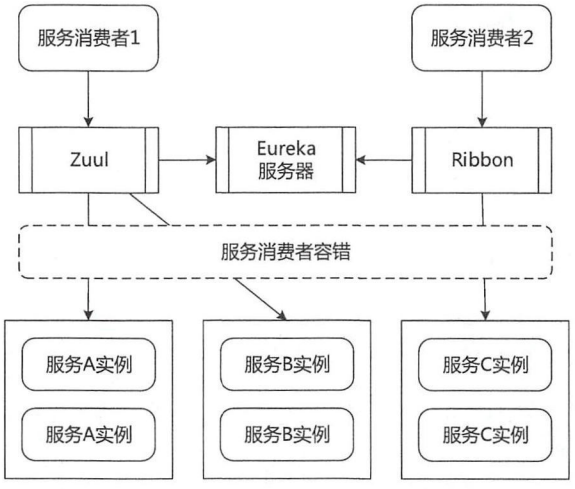


图 6-1　添加 API 网关之后的系统架构图

仅供非商业用途或交流学习使用

本章将使用 Spring Cloud Netflix Zuul 组件来实现 API 网关。与负载均衡器一样，前置的服务网关同样可能面临成为系统瓶颈的风险，所以服务网关本身也需要具备高伸缩性。在设计上，一方面要确保网关本身的无状态性，即网关不应该保存任何与服务调用相关的数据。另一方面网关本身也应该是非常轻量级的，不应该把复杂的业务操作放在网关中。

6.1 服务网关的设计理念

在本章开篇，我们简要介绍了服务网关产生的原因和基本需求。本节将从服务网关的作用以及它所应该具备的结构和功能角度出发，对服务网关的设计理念做进一步阐述。

6.1.1 服务网关的作用

API 网关的作用主要体现在以下几个方面。

（1）解耦

API 网关使客户端和服务器端在调用关系和部署环境上进行解耦，向客户端隐藏了应用如何被划分到微服务的细节。尽管微服务架构支持客户端直接与微服务交互的方式，但当需要交互的微服务数量较多时，解耦就成为一项核心需求。

（2）API 优化

API 网关为每个客户端提供最优的 API。在多客户端场景中，对于同一个业务功能，不同的客户端一般需要不同的数据。例如，对于同一个用户的查询功能，PC 端会一次性获取所有数据，而对于手机 APP 而言，可以通过分页的方式逐步加载用户信息。

（3）简化调用过程

由于能够对返回数据进行灵活处理，API 网关减少了请求往返次数，从而简化了客户端的调用，也提高了服务访问的性能。比如，API 使得客户端可以在一次请求中向多个服务提取数据。请求数量的减少也会直接提升用户体验。

另外，API 网关也增加了系统的复杂性和响应时间，因为 API 网关作为单独的应用程序也需要进行开发、部署和管理。为了暴露每个微服务的 HTTP 端点，开发人员必须更新 API 网关。同时，通过 API 网关访问服务也间接多了一层网络跳转。但是我们认为 API 网关模式的缺点相比其优点是微不足道的，因此该模式目前被广泛应用在微服务架构设计和实现中。如图 6-2 所示，API 网关作为单一入口通过对请求进行转换和适配完成与后台微服务体系的整合，面向各种客户端提供统一服务。

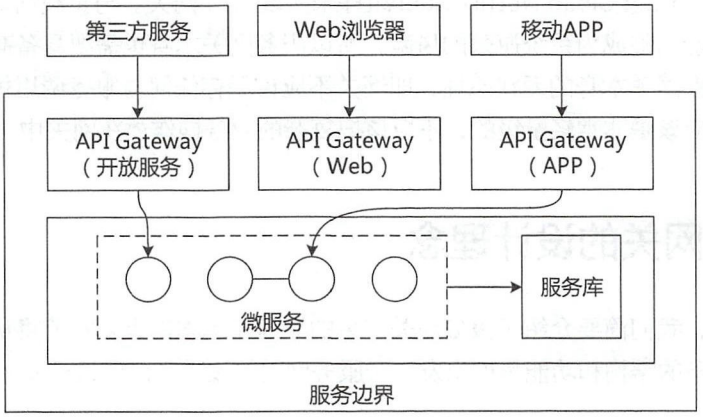


图 6-2　API 网关模式示意图

6.1.2　服务网关的结构和功能

API 网关的功能结构如图 6-3 所示。在这个结构背后，我们需要挖掘其所应具备的核心功能，从而发挥上文中提到的各项作用。

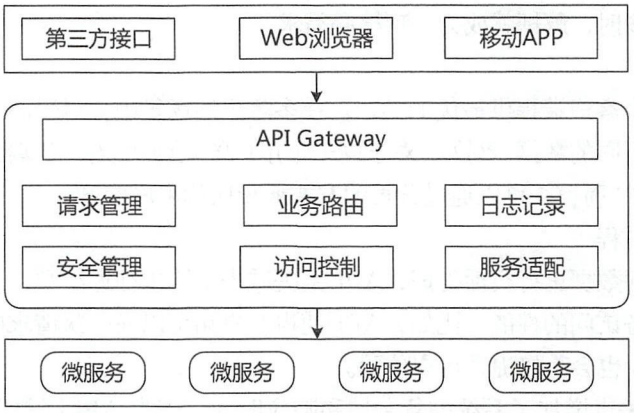


图 6-3　API 网关功能结构

API 网关在实现上需要提供如下功能。

（1）业务路由支持

业务路由支持是 API 网关的核心功能，我们可以在网关层制定灵活的路由策略。针对一些特定的 API，我们需要设置白名单、路由规则等各类限制。更常见的情况，我们需要对网关背后的各种微服务进行统一管理，从而为外部请求提供更加简单高效的路由服务。关于业务功能之间的整合关系以及变更都可以在网关层单独操作，这时候网关表现为是一种门面（Facade）

功能。

（2）报文格式转换

API 网关的一大作用在于构建异构系统，如图 6-4 所示，API 网关作为单一入口通过协议转换整合后台基于 REST、AMQP 和 Dubbo 等不同风格和实现技术的微服务，为面向 Web、Mobile、Open 等特定客户端提供统一服务。

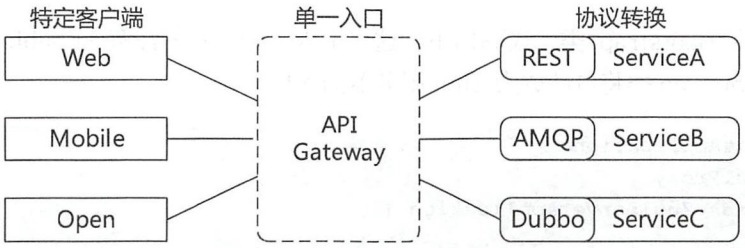


图 6-4　API 网关与异构系统

（3）安全性控制

一般而言，无论是内网还是外网的接口都需要执行用户身份认证，而用户认证在一些规模较大的系统都会采用统一的单点登录（Single Sign On，SSO）功能。如果每个微服务都要对接单点登录系统，显然比较浪费资源且开发效率低下。API 网关是统一管理安全性的绝佳场所，可以将认证的部分抽取到网关层，然后各个微服务系统无须关注认证的逻辑而只关注自身业务即可。常见的安全性技术如秘钥交换、客户端认证与报文加解密等功能都可以在 API 网关中加以实现。

（4）访问控制

某些场景下需要控制客户端的访问次数和访问频率，对于一些高并发系统有时还会有限流的需求。为了防止站点不被未知的大流量冲跨，在网关上可以配置一个阈值，当请求数超过阈值时就直接返回错误而不继续访问后台服务。

6.2　使用 Zuul 构建服务网关

本节内容将详细介绍使用 Zuul 构建服务网关的方法，我们会发现通过 Zuul 构建网关服务器的过程非常简单，而基于 Zuul 内置的配置功能完成服务路由的灵活映射同样也不复杂。

6.2.1　构建 Zuul 服务器

与其他微服务一样，服务网关本身也是一种服务，它是一个标准的 Spring Boot 应用程序。

为了构建 Zuul 服务器，下面创建一个新的 maven 工程 zuul-server 并引入 spring-cloud-starter-zuul 依赖，如下所示。

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
```

然后再创建 Bootstrap 类，代码如下，这里引入了一个新的注解@EnableZuulProxy，嵌入该注解的 Bootstrap 类将自动成为 Zuul 服务器的入口。

```
@SpringBootApplication
@EnableZuulProxy
public class ZuulServerApplication {

    public static void main(String[] args) {
       SpringApplication.run(ZuulServerApplication.class, args);
    }
}
```

@EnableZuulProxy 注解非常强大，基于该注解就可以使用 Zuul 中的各种内置过滤器实现复杂的服务路由，关于 Zuul 提供的过滤器机制将在本章 6.3 节中介绍。

服务网关的定位和功能势必涉及服务的发现和调用，所以服务网关与注册中心及负载均衡关系密切。一方面，以 Eureka 为代表的注册中心为服务路由提供了服务定义信息，这是能够实现服务路由的基础。另一方面，服务网关在服务的消费者和提供者之间提供了一层反向代理，充当着前置负载均衡器的角色。服务网关的定位决定了 Zuul 需要依赖于 Eureka 和 Ribbon。

为了与 Eureka 进行交互，在 zuul-server 中，我们需要在 application.yml 配置文件中添加对 Eureka 的集成，配置内容如下所示，关于各个配置项的内容已在第 3 章中做了详细介绍，这里不再赘述。

```
server:
  port: 5555

eureka:
  instance:
    preferIpAddress: true
  client:
    registerWithEureka: true
    fetchRegistry: true
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

Zuul可以与Ribbon进行集成，同时，在Zuul中还可以基于Hystrix设置服务容错相关的运行时参数。这部分内容将在下一节中具体展开讨论。

在Spring Cloud Netflix Zuul中，还有一个与@EnableZuulProxy类似的注解@EnableZuulServer，我们可以把@EnableZuulProxy简单理解为@EnableZuulServer的增强版，当Zuul与Eureka、Ribbon等组件配合使用时，我们使用的是@EnableZuulProxy注解。@EnableZuulServer注解提供的内置过滤器并不是创建一个反向代理服务器或者尝试与Eureka进行集成。当我们使用其他注册中心（如在第3章中提到的Consul、Etcd等组件）或者想要自建路由机制时可以使用@EnableZuulServer注解，因为本书是基于Spring Cloud整套技术框架来搭建微服务架构体系，所以不对该注解做进一步展开。

依次启动Eureka服务和Zuul服务，然后访问http://localhost:5555/routes就会发现一些键-值对信息，这些键-值对信息实际上就是服务路由映射信息。http://localhost:5555/routes是Zuul提供的服务路由端点，展示了目前在Zuul中配置的服务路由信息。至于这些服务路由信息如何进行配置和管理就是下一节需要讨论的内容。

6.2.2 配置Zuul服务路由

正如我们所想象的API网关，Zuul为我们实现了动态路由、监控、安全等边缘服务，并提供了@EnableZuulProxy注解支持代码中嵌入Zuul反向代理服务器。对于API网关而言，最重要的功能就是服务路由，即通过Zuul访问的请求会路由并转发到对应的后端服务中。通过Zuul进行服务访问的URL示例如下所示，其中，zuulservice代表Zuul服务器地址，而解析斜体的*orderservice*所对应的后端服务就需要依赖于位于Zuul中的服务路由信息。下文中，我们把*orderservice*称为目标服务。

```
http://zuulservice:5555/orderservice/v1/orders/001
```

通常，Zuul是通过基于注册中心的服务发现机制实现自动化的服务路由功能，但同样也可以采用手工设置静态服务路由的方式。本节将分别对这两种服务路由的实现方式做详细介绍。

1. 通过注册中心映射服务路由

通过注册中心映射服务路由依赖于Eureka，一般也存在两种方式，一种是系统自动映射，一种是通过配置进行映射。无论是哪种方式，其基本的原理都是基于目标服务找到Eureka中的服务定义，然后再通过Zuul进行服务调用。图6-5展示了这一过程，目标服务orderservice最终会路由到某一个具体的Order服务实例。

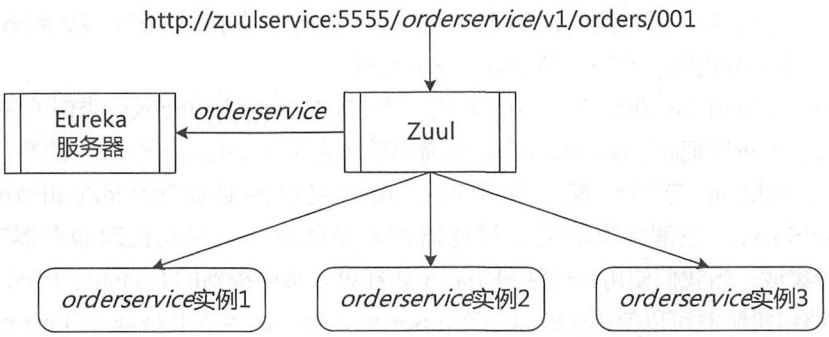


图 6-5　通过注册中心映射服务路由示意图

（1）系统自动映射

系统自动映射最简单，从开发角度讲我们不需要做任何事情。因为在 Eureka 中已经保存了各种服务定义信息，而服务定义信息中包含了各个服务的名称，所以 Zuul 就可以把这些服务的名称与目标服务进行自动匹配，匹配的规则就是直接将目标服务映射到服务名称上即可。例如，在 Order 服务中的 bootstrap.yml 文件中通过以下配置项指定了该服务的名称为 orderservice，那么就可以通过 http://zuulservice:5555/orderservice/v1/orders/001 访问到该服务，注意到该 URL 中的目标服务就是 orderservice，与服务定义中的名称保持一致。

```
spring:
   application:
      name: orderservice
```

Zuul 会从 Eureka 中获取当前所有已注册的服务，然后自动生成服务名称与目标服务之间的映射关系。当我们把案例中前几章介绍的 Product 服务、Account 服务、Order 服务以及下一章要介绍的 config-server 配置中心服务全部注册到 Eureka 中时，访问 http://localhost:5555/routes 端点时可以得到如下路由信息，其中“/orderservice/**”中后半部分的“**”代表所有访问该路径及子路径的请求都将被自动路由到注册在 Eureka 中的名为 orderservice 的某一个服务实例。

```
{
    "/orderservice/**":"orderservice",
    "/productservice/**":"productservice",
    "/accountservice/**":"accountservice",
    "/configserver/**":"configserver"
}
```

如果注册一个新服务或下线某个已有服务，那么这份映射列表也会做相应的调整，这个过程对于开发人员完全透明。

（2）通过配置进行映射

系统自动映射非常方便，但也有明显的限制。从服务网关的定位和作用上讲，我们希望服务路由能够更加灵活和强大。对服务映射关系有更多的定制化需求是很常见的，比方说不使用默认的服务名称来命名目标服务，或者在各个请求路径之前加一个统一的前缀（Prefix）等。Zuul 充分考虑到了这些需求，开发和运维人员可以通过配置实现服务路由的灵活映射。

首先可以在 zuul-server 工程下的 application.yml 配置文件中通过如下方式为 Product 服务、Account 服务和 Order 服务这 3 个服务配置特定的服务名称与请求地址之间的映射关系，可以看到我们分别使用/order、/product、/account 来指定请求根地址。以 orderservice 为例，现在访问 http://zuulservice:5555/order/v1/orders/001 就相当于将请求发送给了 Eureka 中的 orderservice 实例。

```
zuul:
   routes:
     orderservice: /order/**
     productservice: /product/**
     accountservice: /account/**
```

现在重启 zuul-server 并访问 http://localhost:5555/routes 端点，得到的服务路由映射关系就变成如下的结果。可以看到在原有路由信息的基础上，Zuul 生成了 3 条新的路由信息分别对应于配置文件中的配置信息。

```
{
    "/order/**":"orderservice",
    "/product/**":"productservice",
    "/account/**":"accountservice",
    "/orderservice/**":"orderservice",
    "/productservice/**":"productservice",
    "/accountservice/**":"accountservice",
    "/configserver/**":"configserver"
}
```

基于以上配置，相当于访问某一个服务的入口变成了两个，一个是系统自动映射的路由，一个是通过配置所生成的路由。有时候，我们可能不希望系统自动映射的路由被外部所使用，可以通过 ignored-services 配置项把它们从服务路由中去掉。以 orderservice 为例，ignored-services 配置项的使用方法如下所示。

```
zuul:
   routes:
     ignored-services: 'orderservice'
     orderservice: /order/**
```

当通过 ignored-services 配置项去掉所有系统自动映射的路由关系，服务路由信息就是如下所示的结果。想要去掉所有系统自动映射的路由关系也可以直接使用 ignored-services: '*' 配置项。

```
{
    "/order/**":"orderservice",
    "/product/**":"productservice",
    "/account/**":"accountservice",
}
```

另一个比较常见的需求场景是希望在各个服务请求地址中添加一个前缀。例如，在一个大型的微服务架构中，可能会有非常多的微服务，这就需要对这些服务进行全局性的规划，通过模块或子系统的方式进行管理。在各个服务请求地址上添加一个前缀用来标识模块和子系统是一项最佳实践。这时候就可以用到 Zuul 提供的“prefix”配置项，示例如下所示。将“prefix”配置项设置为“/order”，代表所有配置的路由请求地址之前都会自动添加“/order”前缀。

```
zuul:
   prefix:  /order
   routes:
     ignored-services: '*'
     orderservice: /order/**
     productservice: /product/**
     accountservice: /account/**
```

现在访问 http://localhost:5555/routes 端点可以得到如下路由信息，可以看到所配置的前缀已经生效。

```
{
    "/order/order/**":"orderservice",
    "/order/product/**":"productservice",
    "/order/account/**":"accountservice",
}
```

2. 通过手工设置静态服务路由

一般场景下，通过注册中心映射服务路由已经可以支持日常的开发工作。但是 Zuul 还提供了不依赖于 Eureka 的服务路由方式，这种方式可以让 Zuul 具备更多的扩展性，因为我们可以使用自定义的路由规则完成与其他各种第三方系统的集成。我们知道 Spring Cloud 中各个服务之间都是通过轻量级的 HTTP 协议进行交互，而 HTTP 协议具备技术平台无关性，只要能够获取服务的 HTTP 端口地址，原则上就可以进行远程访问，而不用关注该服务在实现上是采用了何种技术和工具。例如，可以在配置文件中添加如下的静态路由配置，这样访问

“/myusers/**”时，Zuul会将该请求转发到外部的第三方服务http://example.com/users_service。

```
zuul:
  routes:
    users:
      path: /myusers/**
      url: http://example.com/users_service
```

现在的服务路由信息就会变成如下结果。关于静态服务路由，Zuul官方文件给出了几个有用的示例，读者可以自行参考。

```
{
    "/order/myusers/**":"http://example.com/users_service",
    "/order/order/**":"orderservice",
    "/order/product/**":"productservice",
    "/order/account/**":"accountservice",
}
```

手工设置静态服务路由的方式同样可以集成Ribbon，具体方式如下所示。配置一个userservice路由信息，通过“/userstatic/**”映射到serviceId为“userservice”的服务中。然后我们希望通过自定义Ribbon的方式来实现客户端负载均衡，这时候需要关闭Ribbon与Eureka之间的关联，可以通过“ribbon.eureka.enabled: false”配置项完成这一目标。在不使用Eureka的情况下，需要手工指定Ribbon的服务列表。“users.ribbon.listOfServers”配置项提供了这方面的支持，如在下面的示例中“http://userservice1:8080, http://userservice2:8080”就为Ribbon提供了两个服务定义作为实现负载均衡的服务列表。

```
zuul:
  routes:
    userservice:
      path: /userstatic/**
      serviceId: userservice

ribbon:
  eureka:
    enabled: false

userservice:
  ribbon:
    listOfServers: http://userservice1:8080,http://userservice2:8080
```

最后，我们还有必要提一下Zuul中集成Hystrix的方法，常见的场景就是设置服务访问的超时时间。可以在配置文件中添加如下配置项完成对经由Zuul的所有服务访问的Hystrix超时

时间的设置。例如，在 zuul-server 的 application.yml 文件中添加如下配置信息的效果就是覆写 Hystrix 的默认超时时间为 3000 毫秒。

```
hystrix:
  command:
    default:
       execution:
          isolation:
             thread:
                timeoutInMilliseconds: 3000
```

以上配置项对 Zuul 中的所有服务均生效，如果想要设置具体的某一个服务（orderservice）的 Hystrix 超时时间，把“hystrix.command.default”段改为“hystrix.command.orderservice”即可。

6.3 Zuul 基本架构

Zuul 提供的核心功能是服务路由，关于服务路由的实现机制我们可能没有明确的思路。但任何经由 Zuul 的服务访问都会经历从请求到响应的工作流程，这点与基于 Web 的请求响应过程非常相似。在以 Servlet 为代表的 Web 应用中，服务的请求响应过程是管道-过滤器（Pipe-Filter）模式的典型应用。而在 Zuul 中，我们会发现过滤器同样发挥着根本性的作用。本节内容将对 Zuul 的基本架构做详细介绍，特别是它所具备的过滤器机制。

6.3.1 ZuulFilter 组件架构

ZuulFilter 是 Zuul 中的关键组件，在介绍 ZuulFilter 之前，先来回顾一下管道-过滤器模式。

1. 管道-过滤器模式回顾

管道-过滤器结构主要包括过滤器和管道两种元素（见图 6-6）。

- 过滤器（Filter）

过滤器是一种功能组件，负责对数据进行加工处理。每个过滤器都有一组输入端口和输出端口，从输入端口接收数据，经过内部加工处理之后，传送到输出端口上。

- 管道（Pipe）

数据通过相邻过滤器之间的管道进行传输，管道可以看作输入数据流和输出数据流之间的通路。

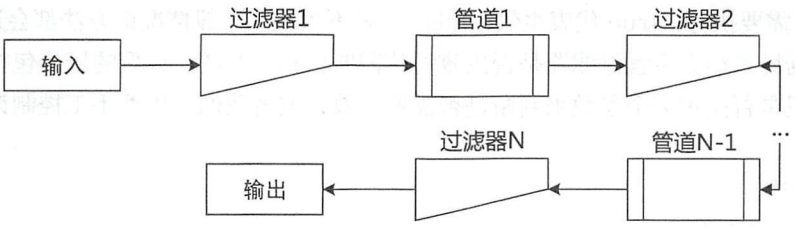


图 6-6 管道-过滤器风格

管道-过滤器结构将数据流处理分为几个顺序的步骤来执行，一个步骤的输出是下一个步骤的输入，每个处理步骤均由一个过滤器来实现。每个过滤器独立完成自己的任务，不同过滤器之间不需要进行交互。在管道-过滤器结构中，数据输出的最终结果与各个过滤器的执行顺序无关。这些特性允许将系统的输入和输出看作是各个过滤器行为的简单组合，独立的过滤器能够降低组件之间的耦合程度，我们也可以很容易地将新过滤器添加到现有的系统之中，原有过滤器也可以很方便地被改进的过滤器所替换以扩展系统的业务处理能力。

图 6-7 是以 Nginx、Tomcat 为代表的 Web 应用服务器对 HTTP 请求的处理流程，可以看到 Web 应用服务器对请求的响应过程就是典型的管道-过滤器模式的体现。当请求到达服务器时，服务器分配资源并根据配置获取对应的处理器进行处理，处理结果再经过层层过滤产生最终响应，我们可以在过滤链中根据需要添加过滤器以确保响应结果满足业务需求。

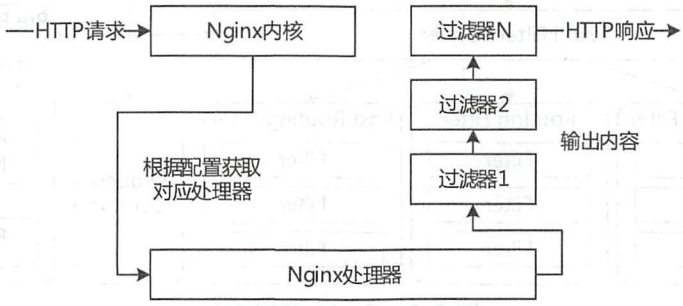


图 6-7 Nginx HTTP 模块处理流程

Zuul 响应 HTTP 请求的过程与图 6-7 类似，内部提供了 ZuulFilter 组件来实现过滤器机制。

2. ZuulFilter 组件

基于 ZuulFilter 组件，通过继承该抽象类，覆写几个关键方法就能达到自定义调度请求的效果。ZuulFilter 定义了以下几个主要方法。

- filterOrder()方法

Filter 执行顺序，通过数字指定。数字越大，优先级越低。

- shouldFilter()方法

Filter 是否需要执行，true 代表生效，false 代表不生效。一般情况该方法都会返回 true，当我们需要根据场景动态设置过滤器是否生效时就需要用到该方法。典型的场景包括可能需要根据服务请求里是否携带某个参数来判断是否需要生效，或者我们希望能手工控制该过滤器是否生效。

- run()方法

Filter 具体实现逻辑。

- filterType()方法

Filter 类型，内置过滤器分为 PRE、ROUTING、POST 和 ERROR 4 种，本节后续内容会具体展开介绍。

Zuul 提供了一个框架，可以对过滤器进行动态的加载、编译和运行。过滤器之间没有直接的交互，而是通过一个 RequestContext 静态类来进行数据传递。RequestContext 类中用 ThreadLocal 变量来记录每个 Request 所需要传递的数据。Zuul 过滤器整体架构如图 6-8 所示。

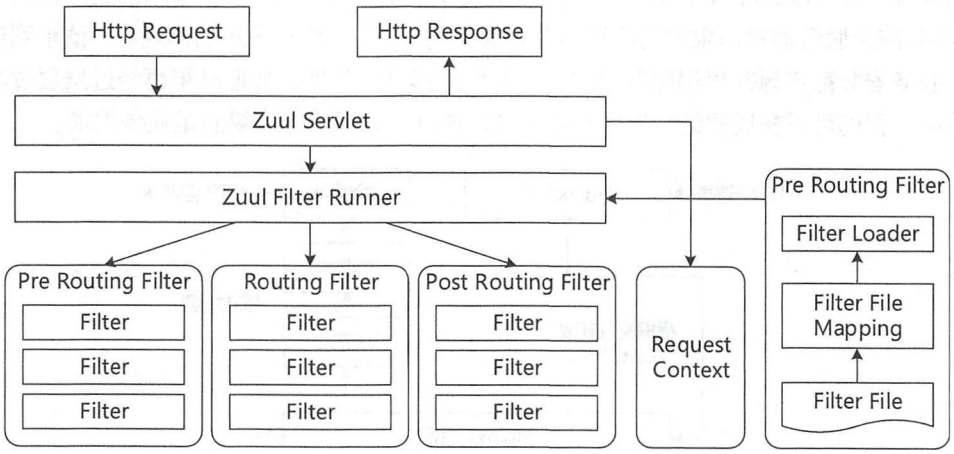


图 6-8　Zuul 过滤器架构

Zuul 中定义了 4 种标准过滤器类型，这些过滤器类型对应于一个请求到达 API 网关直到返回响应结果的典型生命周期。

- PRE

这种过滤器在请求到达目标服务器之前调用，如身份验证、在集群中选择目标服务器、记录请求日志等。

- ROUTING

在这种过滤器中把用户请求发送给目标服务器。发送给目标服务器的用户请求在这类过滤器中被重新构建，并使用 Apache HttpClient 或者 Netfilx Ribbon 向目标服务器发起调用。

- POST

这种过滤器在用户请求从目标服务器返回以后执行，可以在返回的响应中添加响应头信息以及做各种统计等。

- ERROR

在其他阶段发生错误时执行该过滤器。

Zuul 请求的生命周期如图 6-9 所示，该图详细描述了各种类型过滤器的执行顺序。

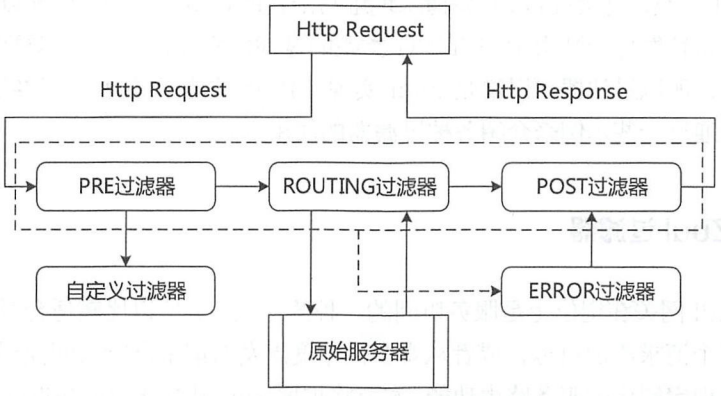


图 6-9　Zuul 过滤器生命周期

Spring Cloud 默认为 Zuul 编写并启用了很多内置的过滤器，这里列举部分常见的过滤器及其功能。

- ServletDetectionFilter

PRE 类过滤器，用于检查请求是否是通过 Spring Dispatcher 转发。

- FormBodyWrapperFilter

PRE 类过滤器，解析表单数据，并为请求重新编码。

- PreDecorationFilter

PRE 类过滤器，提供 RouteLocator 确定在哪里和如何路由。

- DebugFilter

PRE 类过滤器，调试用的过滤器。

- SendForwardFilter

ROUTING 类过滤器，使用 Servlet RequestDispatcher 转发请求。

- RibbonRoutingFilter

ROUTING 类过滤器，使用 Ribbon、Hystrix 和可插拔的 HTTP 客户端发送请求。

- SimpleHostRoutingFilter

ROUTING 类过滤器，通过 Apache HttpClient 向指定的 URL 发送请求。

- SendResponseFilter

POST 类过滤器，将 Zuul 所代理的服务请求响应结果写入当前响应。

- SendErrorFilter

ERROR 类过滤器，处理有错误的请求响应，默认转发到/error 端点。

我们介绍 Zuul 的方式是先讲服务路由，再讲过滤机制。从表面上讲，路由功能负责将请求转发到具体的微服务实例上，而过滤器功能则是对请求的处理过程进行干预。但事实上，路由功能在运行过程中，Zuul 的路由映射和请求转发都是由几个内置的过滤器实现的。其中，路由映射主要通过 PRE 类过滤器完成，它将请求地址与配置的路由规则进行匹配以找到需要转发的目标地址。而请求转发部分则由 ROUTING 类型的过滤器完成，对 PRE 类型过滤器获取的路由地址进行转发，所以过滤器可以说是 Zuul 实现 API 网关功能最为核心的组件。在本章后续内容中，我们将通过一些示例来介绍各种过滤器的使用方式。

6.3.2 使用 Zuul 过滤器

我们知道 API 网关在定位上是服务访问的一种统一入口，所以比较适合对请求做各种拦截和处理，如对每个请求添加日志，或者从安全性角度出发对请求进行访问控制等，当然我们也可以根据需要实现定制化的服务路由功能。本节将扩展 Zuul 内置的 ZuulFilter 组件以实现上述需求。

1. 使用 PRE/POST 过滤器

我们先来考虑一个最简单的应用场景，即对每个经由 Zuul 的请求做简单的日志记录。对于每一个具体的应用场景，首先需要分析应该采用哪种合适的过滤器类型。我们知道 ZuulFilter 中存在 PRE、ROUTE、POST 和 ERROR 4 种过滤器类型，对于日志记录需求而言，原则上任何一种过滤器都能实现这一需求。一般而言，把日志记录放在 PRE 类过滤器中相对更为合适。

上一节中提到 Zuul 内置的 ZuulFilter 类只有 4 个方法，分别是 filterOrder()、shouldFilter()、run()和 filterType()。同时，我们也知道在 ZuulFilter 架构体系中存在一个 RequestContext 的静态类，可以通过该静态类来进行数据传递。所以实现一个过滤器的通用做法就是扩展 Zuul 内置的 ZuulFilter 类，覆写 ZuulFilter 类的 4 个方法并通过 RequestContext 传递数据。

以下代码展示了扩展 ZuulFilter 以实现日志记录的 RequestLogFilter 类的代码结构，其中，filterOrder()、shouldFilter()和 filterType()方法非常简单，我们设置该过滤器的类型为“pre”，让 Zuul 知道它是一个 PRE 类的过滤器。而在 run()方法中可以使用 RequestContext.getCurrentContext()静态方法获取当前请求的 RequestContext，然后通过 RequestContext 进一步获取请求地址。整个过程因为不需要提供任何数据给下一个过滤器，所以 run()方法简单返回 null 即可。

```
@Component
public class RequestLogFilter extends ZuulFilter{
    private static final Logger logger =
        LoggerFactory.getLogger(RequestLogFilter.class);

    @Override
    public String filterType() {
        return "pre";
    }

    @Override
    public int filterOrder() {
        return 1;
    }

    public boolean shouldFilter() {
        return true;
    }

    public Object run() {

        RequestContext ctx = RequestContext.getCurrentContext();
        logger.debug(String.format("Processing incoming request for
              {}.",  ctx.getRequest().getRequestURI()));

        return null;
    }
}
```

实现POST过滤器的方法与上面的例子完全一致，如下所示即为一个POST过滤器的实现示例。我们把filterType设置为“post”，然后重写run()方法。可以看到该方法从RequestContext中获取“serviceId”，也就是当前的服务名称，然后自定义一个HTTP头，并把服务名称添加到Response的HTTP头中。

```
@Component
public class ResponseServiceNameFilter extends ZuulFilter{
    private static final Logger logger =
         LoggerFactory.getLogger(ResponseServiceNameFilter.class);

    @Override
    public String filterType() {
        return "post";
    }
```

```
    @Override
    public int filterOrder() {
        return 1;
    }

    public boolean shouldFilter() {
        return true;
    }

    public Object run() {
        RequestContext ctx = RequestContext.getCurrentContext();
        if (ctx.get("serviceId") != null) {
          String serviceName = ctx.get("serviceId").toString();
          logger.debug(String.format("Current service is {}.",
             serviceName));

          ctx.getResponse().addHeader("ServiceName", serviceName);
        }

        return null;
    }
}
```

使用 PRE 或 POST 过滤器可以实现非常灵活的请求处理。当然，Zuul 的核心功能是服务路由，接下来介绍如何使用 ROUTE 过滤器来实现自定义的服务路由功能。

2. 使用 ROUTE 过滤器实现自定义服务路由

在服务发布过程中，我们有时候会使用所谓的“灰度发布”（Gray Release）机制，从而对部分新功能进行小规模的试用。灰度发布是指在服务新旧版本之间能够平滑过渡的一种发布方式。实现灰度发布的方式有很多，这里以一种简化的方式演示 ROUTE 过滤器的使用方式。

当发送请求到指定服务时，我们可以在请求中指定一个“greyFlag”参数用于表明该请求将执行灰度操作。在上一节的示例中演示了通过 RequestContext 获取当前服务名称的方法，我们可以在服务名称前简单添加“grey_”前缀用来表明灰度服务的名称。这样一旦捕获到带有“greyFlag”参数的服务请求，ROUTE 过滤器就将服务请求转发到灰度服务，从而实现服务路由的转换。灰度发布路由的工作流程如图 6-10 所示。

基于以上设计思想，我们创建一个新的 GreyReleaseRouteFilter 过滤器类，并把它的 filterType 设置为“route”。然后根据灰度发布的需求提供如下的 run()方法（由于篇幅有限，这里仅截取部分核心代码作为示例）。

仅供非商业用途或交流学习使用

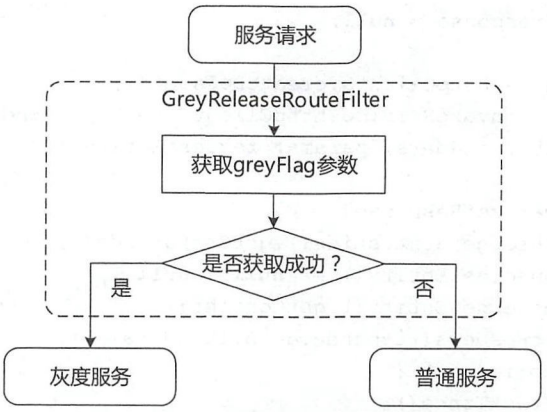


图6-10 灰度发布路由示意图

```
@Override
public Object run() {
    RequestContext ctx = RequestContext.getCurrentContext();
    HttpServletRequest request = ctx.getRequest();

    String greyFlag = request.getParameter("greyFlag");
    if (greyFlag == null)
        return null;

    if (ctx.get("serviceId") == null)
        return null;

    //获取灰度环境服务地址
    String serviceName = ctx.get("serviceId").toString();
    String greyServiceName = "grey_" + serviceName;
    String originalEndpoint = ctx.getRequest().getRequestURI();
    String newEndpoint = originalEndpoint.replace(serviceName,
         greyServiceName);

    //准备服务调用参数
    MultiValueMap<String, String> headers =
        this.helper.buildZuulRequestHeaders(request);
    MultiValueMap<String, String> params =
        this.helper.buildZuulRequestQueryParams(request);
    String verb = request.getMethod().toUpperCase();
    InputStream requestEntity = getRequestBody(request);
    this.helper.addIgnoredHeaders();

    //发起服务调用
    HttpClient httpClient = null;
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
    HttpResponse response = null;
    try {
        httpClient = HttpClients.createDefault();
        response = invokeService(httpClient, verb, newEndpoint,
            request, headers, params, requestEntity);

        this.helper.setResponse(
            response.getStatusLine().getStatusCode(),
            response.getEntity() == null ? null :
            response.getEntity().getContent(),
            revertHeaders(response.getAllHeaders()));
    } catch (Exception ex) {
        ex.printStackTrace();

    } finally {
        try {
            httpClient.close();
        } catch (IOException ex) {
        }
    }

    return null;
}
```

以上方法通过获取灰度环境服务地址、准备服务调用参数、发起服务调用等步骤实现了服务路由，其中，在“准备服务调用参数”步骤中依赖 Zuul 提供的 ProxyRequestHelper 工具类。而在“发起服务调用”步骤中，使用 HttpClient 工具类完成远程 HTTP 请求的发送和响应，涉及的 invokeService()方法代码如下所示，这里基本上就是对整个 HTTP 请求过程的封装。

```
private HttpResponse invokeService(HttpClient httpclient, String verb,
    String uri, HttpServletRequest request, MultiValueMap<String,
    String> headers, MultiValueMap<String, String> params, InputStream
    requestEntity) throws Exception {

    URL host = new URL(uri);
    HttpHost httpHost = new HttpHost(host.getHost(), host.getPort(),
        host.getProtocol());

    HttpRequest httpRequest;
    int contentLength = request.getContentLength();
    InputStreamEntity entity = new InputStreamEntity(requestEntity,
        contentLength, request.getContentType() != null ?
        ContentType.create(request.getContentType()) : null);
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
        HttpPost httpPost = new HttpPost(uri);
        httpRequest = httpPost;
        httpPost.setEntity(entity);
        httpRequest.setHeaders(convertHeaders(headers));

        HttpResponse httpResponse = httpclient.execute(httpHost,
                httpRequest);
        return httpResponse;
    }
```

通过以上示例，我们基于ROUTE过滤器可以实现灵活而强大的自定义服务路由机制。该示例内容上并不具备通用性，但其背后的设计思想和实现过程具有通用的参考价值。

6.4 本章小结

API网关在微服务架构中具有重要地位，它是连接各个服务之间的一种桥梁。本章首先对服务网关的作用、结构及核心功能进行了分析。

Spring Cloud家族中提供了一个专门的API网关实现工具Spring Cloud Netflix Zuul。Zuul同样是一个微服务，需要构建独立的Zuul服务器。而我们使用Zuul服务的主要方式是配置各种服务路由。服务路由是Zuul的核心功能，在使用上有两种主要的方式，一种是通过注册中心自动映射服务路由，另一种是通过手工设置静态服务路由。

本章最后对Zuul的基本架构进行了展开，Zuul的核心组件是ZuulFilter，而ZuulFilter也是管道-过滤器模式的一种具体实现。ZuulFilter组件中内置了多种不同的过滤器类型，我们可以使用它的可扩展架构开发自定义过滤器，从而实现对服务请求的各个阶段进行拦截和处理。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

第 7 章

Spring Cloud Config 与配置中心

配置管理的需求在任何类型的系统中都存在，而且随着业务复杂度的提升和技术架构的演变，系统对配置信息的管理方式也会提出越来越高的要求。例如，在单块系统中，配置管理方式的演变过程是这样的：刚开始时配置文件比较少，更新频率也不会太高，所以倾向于把所有配置跟源代码一起放在代码仓库中；之后由于配置文件数量和改动频率的增加，就会考虑将配置文件从代码仓库中分离出来，或者放在 CI 服务器上通过打包脚本打入应用包中，或者直接放到运行应用的服务器特定目录下，剩下的非文件形式的关键配置则存入数据库中。

上述配置管理的演变过程在单块系统阶段非常常见，在确保配置信息安全性的同时，也往往可以运行得很好。但到了微服务阶段，面对爆发式增长的应用数量和服务器数量，就显得无能为力。为此，在微服务架构中，一般都需要引入配置中心（Configuration Center）的设计思想和相关工具。

本章重点介绍基于 Spring Cloud Config 的分布式配置中心方案，与前面介绍的基于 Netflix OSS 的各种组件不同，Spring Cloud Config 是 Spring Cloud 家族研发的高可用、分布式配置中心。图 7-1 展示了在前面各章的基础上添加配置中心之后的系统架构图，微服务系统中的各个服务都可能会依赖配置中心以完成配置信息的统一管理。

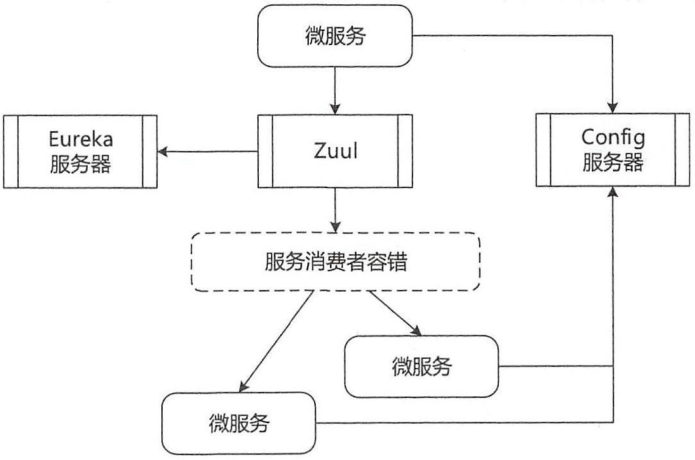


图 7-1　添加配置中心之后的系统架构图

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

7.1 分布式配置中心方案

在具体介绍 Spring Cloud Config 之前，需要先来梳理分布式配置的场景需求，并对目前市面上主流的分布式配置中心实现工具做简要介绍和对比。

7.1.1 分布式配置模型

关于配置信息的管理，常见的做法是把它们存放在配置文件中。在单块系统中，配置文件能够满足需求，围绕配置文件展开的配置管理工作通常不会有太大挑战。但在分布式系统中，配置文件的管理就会暴露出一些问题。在本节中，我们试图梳理分布式配置模型，涉及配置中心在微服务架构中的定位、配置信息的分类和配置管理的需求等各个方面。

1. 配置中心与微服务架构

每一个微服务系统都应该有一个配置中心，图 7-2 展示了微服务系统中配置中心的定位及其基本结构。

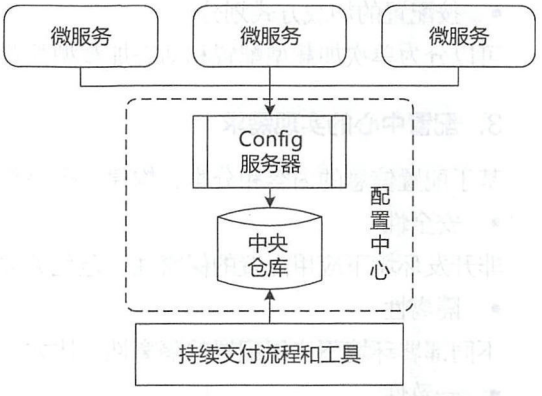


图 7-2 微服务系统中的配置中心基本结构

从图 7-2 中可以看到配置中心由两个核心组件构成，分别是中央仓库和 Config 服务器。

- 中央仓库

配置中心中的所有配置信息都存放在一个中央仓库（Repository）中。中央仓库的实现方式有很多，可以采用 SVN 和 Gitlab 等具备版本控制功能的第三方工具，也可以自建一个具有持久化或内存存储功能的存储媒介。通常，开发人员通过配置管理和持续交付的各项模式和实践并借助于特定工具完成中央仓库中配置信息的更新。关于配置管理和持续交付的相关内容这里不做展开，请参考笔者所著的《微服务原理与架构设计》第 7 章中的相关内容。

- Config 服务器

Config 服务器封装对中央仓库的相关操作，本身也是作为独立的微服务而存在。当需要使用配置信息的微服务启动时，这些微服务会通过 Config 服务器提供的 RESTful 接口获取存储在中央仓库中的所需配置信息。显然，Config 服务器需要屏蔽不同中央仓库实现在技术上的差

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

异。另外，Config 服务器也需要提供一种通知机制，确保中央仓库中的配置信息变化之后能够告知各个微服务，以便各个微服务及时更新本地服务中的配置数据。与第 3 章中介绍的注册中心实现方式类似，这种通知机制可以是轮询机制，也可以是监听机制。

2. 配置信息的分类

微服务架构中的配置信息一般具有如下分类。

- 按配置的来源划分

可分为源代码文件、数据库和远程存储媒介等。

- 按配置的适用环境划分

可分为开发环境、测试环境、预发布环境、生产环境等。

- 按配置的集成阶段划分

可以分为编译时、打包时和运行时。编译时，最常见的有两种，一是源代码级的配置，二是把配置文件和源代码一起提交到代码仓库中。打包时，即在应用打包阶段通过某种方式将配置信息嵌入到最终的应用包中。运行时，是指应用启动前并不知道具体的配置，而是先从本地或者远程获取配置之后再正常启动。

- 按配置的加载方式划分

可以分为单次加载型配置和动态加载型配置。

3. 配置中心的实现需求

基于配置信息的内容和分类，构建一个完整的配置中心至少需要满足如下 4 个核心需求。

- 安全性

非开发环境下应用配置的保密性，避免关键配置发生泄露。

- 隔离性

不同部署环境下应用配置的隔离性，比如非生产环境的配置不能用于生产环境。

- 一致性

同一部署环境下的服务器应用配置的一致性，即所有服务器使用同一份配置。

- 易管理性

分布式环境下应用配置的可管理性，即提供远程管理配置的能力。

采用配置中心也就意味着采用集中式配置管理的设计思想（见图 7-3）。在集中式配置中心中，开发、测试和生产等不同的环境配置信息统一保存在配置中心中，这是一个维度。而另一个维度就是分布式集群环境，需要确保集群中同一类服务的所有服务器保存同一份配置文件并且能够同步更新。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

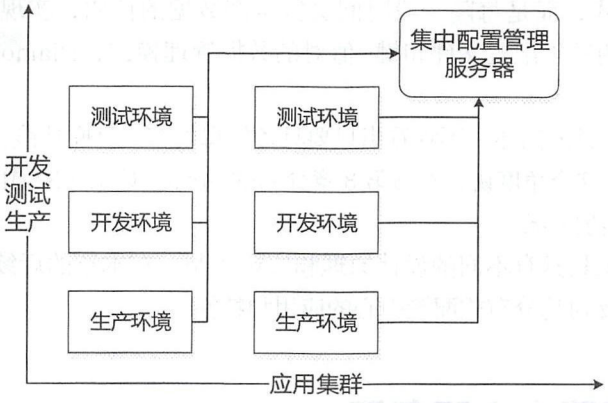


图 7-3　集中式配置中心示例图

7.1.2　配置中心实现工具

基于配置中心的实现需求，业界存在一批典型的分布式配置中心实现工具。这里列举了部分知名的开源实现工具并给出了相应的特性描述。

- Etcd

Etcd 是一个轻量级分布式键-值对数据库，基于 Raft 协议提供了集群环境的服务发现和注册机制。在功能上实现了数据变更监视、目录监听、分布式锁原子操作等特性并通过这些特性来管理节点状态。

- Consul

Consul 是 Google 开源的一个使用 Go 语言开发的服务发现和配置管理中心服务。内置了服务注册与发现框架、分布一致性协议实现、健康检查、键-值对存储、多数据中心方案等诸多特性。在内部实现上，Consul 使用了一种称之为 SWIM 的协议完成分布式环境下的数据计算。

- Zookeeper

Zookeeper 是一种分布式协调工具，目前在分布式配置、分布式锁、集群管理、Leader 选举等领域应用非常广泛。正因为 Zookeeper 是很多分布式工具和框架的基础，本章 7.4 节中会给出 Zookeeper 的相关特性描述，并结合 Zookeeper 和 Spring Cloud Config 做详细对比。

- Disconf

Disconf 是由百度开源的一个分布式配置管理工具，与 Spring 框架有很好的集成，在使用上也提供了友好的 Web 管理界面。从实现机制上讲，Disconf 把配置信息都存储在 MySQL 中，并基于 Zookeeper 提供的监听机制实现数据的实时推送。

- Diamond

Diamond 由阿里巴巴提供，同样把配置数据存储在 MySQL 上，但在获取配置数据时

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

不是使用的推送模式，而是每隔一段时间进行全部数据的拉取，实现过程比较简单。相较 Disconf 同时提供的基于配置文件和键-值对的数据管理模式，Diamond 只支持键-值对结构的数据。

通过以上各个工具的描述，不难看出只要具备信息发布和订阅功能，我们就能够实现一个分布式配置中心。从这个角度讲，本书第 3 章介绍的 Spring Cloud Netflix Eureka 也不失为一种合适的配置中心实现方案。

显然，不同的工具具有不同的设计原理和实现机制，在本章的后续内容中，我们会基于 Spring Cloud Config 讨论分布式配置中心的应用和特性。

7.2 构建配置中心服务器

在图 7-2 中可以看到配置中心有两个核心组件，一个是 Config 服务器，一个是中央仓库。中央仓库的选择可以因地制宜，但 Config 服务器的构建在 Spring Cloud 中只有一种方式。本节将通过引入 Spring Cloud Config 完成 Config 服务器的构建，讨论采用不同类型中央仓库下的配置方案。

7.2.1 引入 Spring Cloud Config

要想构建 Config 服务器，首先需要创建一个新的 maven 工程 config-server 并导入两个组件，分别是 spring-cloud-config-server 和 spring-cloud-starter-config，其中，前者包含了用于构建 Config 服务器的各种组件，相应的 maven 依赖如下所示。

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-config-server</artifactId>
</dependency>

<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

下面将在新建的 config-server 工程中添加一个 Bootstrap 类 ConfigServerApplication，如下所示。

```
@SpringBootApplication
@EnableEurekaClient
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
@EnableConfigServer
public class ConfigServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(ConfigServerApplication.class, args);
    }
}
```

首先，Config 服务本身也是一种微服务，也需要将自身注册到 Eureka 服务器中，所以在 ConfigServerApplication 上也需要添加@EnableEurekaClient 注解，并且在 application.yml 中设置 Eureka 服务器地址。

同时，我们看到在 ConfigServerApplication 中添加了一个崭新的注解@EnableConfig Server。集中式配置服务器的创建通过@EnableConfigServer 注解实现，配置服务器可以将存放在仓库中的配置信息转化为 REST 接口数据供应用服务在分布式环境下使用。

7.2.2 实现基于本地文件系统的配置方案

我们知道中央仓库可以有多种实现方式，Spring Cloud Config 对于中央仓库的处理也是一样，最常见的实现方案就是基于本地文件系统的配置方案和基于 Git 的配置方案。本节先来介绍基于本地文件系统的配置方案，在这种配置方案中，配置信息位于 Config 服务器本地的文件系统中。

为了达到集中化管理的目的，Spring Cloud Config 对配置文件的命名做了约束，使用 label 和 profile 概念指定配置信息的版本及运行环境，其中，label 表示配置版本控制信息，而 profile 中的 dev、prod、test 分别对应着开发、生产和测试环境。配置文件可以使用如下所示的多种命名方式，并同时支持.properties 和.yml 两种文件格式。

```
/{application}.yml
/{application}-{profile}.yml
/{label}/{application}-{profile}.yml
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties
```

在案例中，当使用本地配置文件方案构建 Config 服务器时，一种典型的项目工程结构如图 7-4 所示。我们在 src/main/resources 目录下创建一个 config 文件夹，再在 config 文件夹下分别创建 accountservice、orderservice 和 productservice 3 个子文件夹，请注意这 3 个子文件夹的名称必须与各个服务自身的名称完全一致。然后可以看到这 3 个子文件夹下面都放着一个以服务名称命名的.yml 配置文件。

仅供非商业用途或交流学习使用

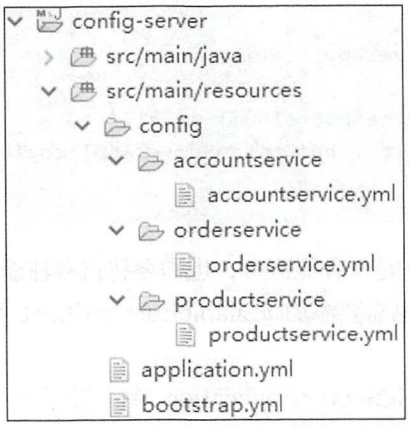


图 7-4　本地配置文件方案下的项目工程结构图

然后，在图 7-4 中的 application.yml 文件中添加如下配置项，通过 searchLocations 指向各个配置文件的路径。

```
server:
   port: 8888
spring:
   profiles:
      active: native
   cloud:
      config:
        server:
           native:
               searchLocations: classpath:config/
                   classpath:config/orderservice,
                   classpath:config/productservice,
                   classpath:config/accountservice
```

现在在 config/productservice/productservice.yml 配置文件中添加如下所示的配置信息，显然这些配置信息用于设置 MySQL 数据库访问的各项参数。

```
spring:
  jpa:
    database: MYSQL
  datasource:
    platform: mysql
    url: jdbc:mysql://127.0.0.1:3306/microservice_product
    username: tianyalan
    password: default_pwd
    driver-class-name: com.mysql.jdbc.Driver
```

仅供非商业用途或交流学习使用

Spring Cloud Config 为用户提供了强大的集成入口，Config 服务器可以将存放在本地文件系统中的配置文件信息自动转化为 RESTful 风格的接口数据。启动 Config 服务器，并访问 http://localhost:8888/productservice 端点时，可以得到如下所示的当前环境配置文件的元数据以及包含在配置文件中的数据库配置信息，其中“profiles”值为“default”，意味着配置文件的 profile 是默认环境，这点从 productservice.yml 文件的命名上就可以看出。而“label”的值是“master”，实际上也是代表着一种默认版本信息。最后的“propertySources”段展示了配置文件的路径以及具体内容。

```
{
    "name":"productservice",
    "profiles":[
        "default"
    ],
    "label": master,
    "propertySources":[
        {
            "name":
                "classpath:config/productservice/productservice.yml",
            "source":{
                "spring.jpa.database":"MYSQL",
                "spring.datasource.platform":"mysql",
                "spring.datasource.url":
                    "jdbc:mysql://127.0.0.1:3306/microservice_product",
                "spring.datasource.username":"root",
                "spring.datasource.password":"default_pwd",
                "spring.datasource
                    .driver-class-name":"com.mysql.jdbc.Driver"
            }
        }
    ]
}
```

图 7-4 中展示的是最简单的一种配置文件管理场景，即没有使用 profile 来区分各个运行环境。在现实中，我们通常会对不同的运行环境设置不同的配置信息，也就会有不同的 HTTP 端点来暴露这些配置信息。带有 profile 的配置文件以及对应的 HTTP 端点映射关系如图 7-5 所示。

假设在 Product 服务中存在一个 test 环境，然后在图 7-4 所示的 productservice 子目录下添加一个 productservice-test.yml 配置文件，该配置文件的配置项与 productservice.yml 完全一致，只是对 MySQL 数据库的 datasource 部分配置项做了调整。那么在 Config 服务器所在的主机上访问 http://localhost:8888/productservice/test，即可得到类似以 JSON 格式表示的配置信息，展示了 test 环境配置文件的元数据以及包含在配置文件中的数据库配置信息。

仅供非商业用途或交流学习使用

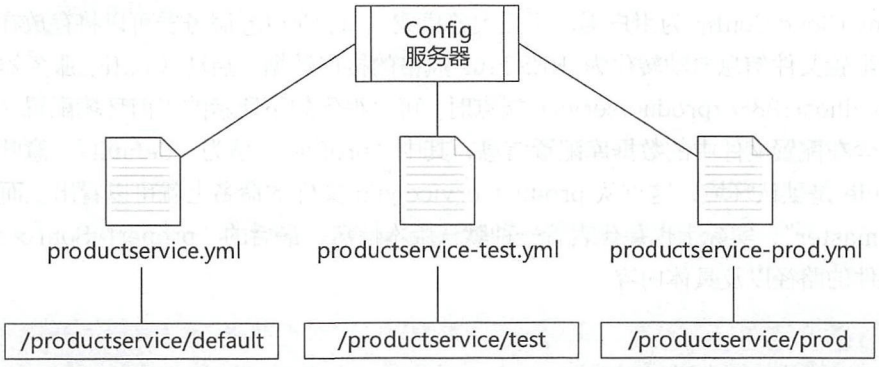


图 7-5　带有 profile 配置文件以及对应的 HTTP 端点映射关系示意图

```
{
    "name":"productservice",
    "profiles":[
        "test"
    ],
    "label":master,
    "propertySources":[
        {
            "name"
            :"classpath:config/productservice/productservice-test.yml",
            "source":{
                "spring.jpa.database":"MYSQL",
                "spring.datasource.platform":"mysql",
                "spring.datasource.url":
                "jdbc:mysql://192.169.0.103:3306/microservice_product",
                "spring.datasource.username":"root",
                "spring.datasource.password":"test_pwd",
                "spring.datasource
                    .driver-class-name":"com.mysql.jdbc.Driver"
            }
        },
        {
            "name"
                :"classpath:config/productservice/productservice.yml",
            "source":{
                "spring.jpa.database":"MYSQL",
                "spring.datasource.platform":"mysql",
                "spring.datasource.url":
                    "jdbc:mysql://127.0.0.1:3306/microservice_product",
                "spring.datasource.username":"root",
                "spring.datasource.password":"default_pwd",
```

仅供非商业用途或交流学习使用


```
                "spring.datasource
                    .driver-class-name":"com.mysql.jdbc.Driver"
            }
        }
    ]
}
```

这里需要注意的是 test 端点会同时返回 default 和 test 两个 profile 对应的所有配置信息。

7.2.3 实现基于 Git 的配置方案

Spring Cloud Config 支持将配置信息放在配置服务器本地的文件系统中，也支持放在远程 Git 仓库中。假如把配置信息放在 Git 仓库中，则 Config 服务器和各个微服务的交互过程如图 7-6 所示。

图 7-6　基于 Git 仓库的服务配置交互过程图

如果使用 Git 作为中央仓库，通常的做法是把所有的配置文件放到自建或公共的 Git 系统中。例如在案例中，可以把各个服务所依赖的配置文件统一放到 Github 上进行托管，如图 7-7 所示。

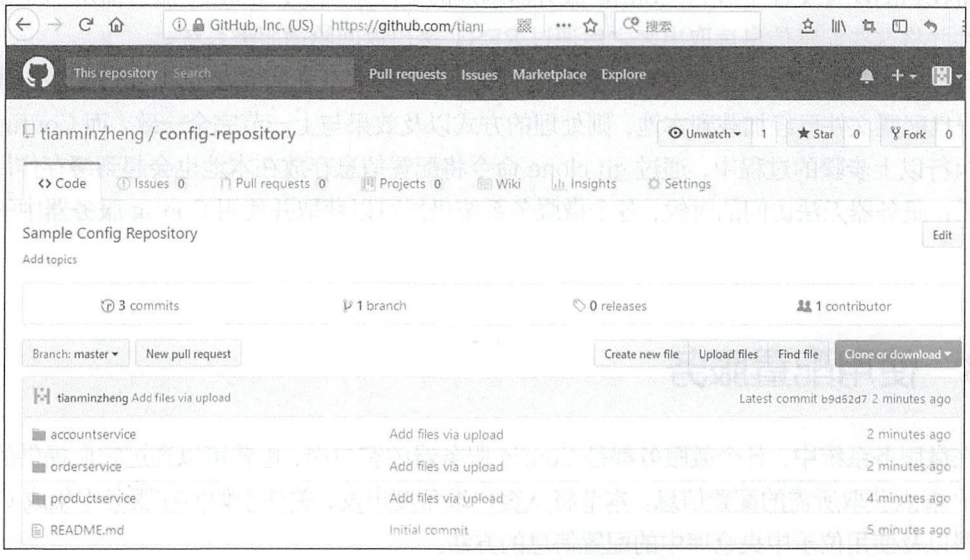


图 7-7　Github 作为中央仓库的效果示意图

仅供非商业用途或交流学习使用

因为改变了中央仓库的实现方式，我们同样需要修改 application.yml，调整后的配置内容如下所示。我们在 spring.cloud.config.server.git 配置段中指定了 Github 相关的各项信息，其中，searchPaths 用于指向各个服务的配置信息所在的目录名称。

```
server:
  port: 8888
spring:
  cloud:
    config:
      discovery:
        enabled: true
      server:
        encrypt.enabled: false
        git:
          uri: https://github.com/tianminzheng/config-repository/
          searchPaths: accountservice,orderservice,productservice
          username: tianyalan
          password: config_pwd
```

基于 Git 的配置方案得到的结果与使用本地文件系统完全一样，但在工作流程上会有所区别。在基于 Git 的配置方案中，各个微服务从 Config 服务器中获取配置信息的工作流程如下：首先各个客户端应用启动时，根据配置的应用名{application}、环境名{profile}和分支名{label}，向 Config 服务器请求配置信息；然后 Config 服务器根据自己维护的 Git 仓库信息和客户端传递过来的配置定位参数去查找具体的配置信息；一旦找到配置信息，则通过 git clone 命令将找到的配置文件下载到 Config 服务器本地的文件系统中；Config 服务器从本地加载配置文件并将这些配置信息读取出来之后通过 REST 接口返回给微服务系统。

可以看到基于 Git 的配置方案的最终结果也是将位于 Git 仓库中的远程配置文件加载到本地，一旦配置文件已经加载到本地，则处理的方式以及效果与上一节完全一致。而 Config 服务器在执行以上步骤的过程中，通过 git clone 命令将配置信息存放在本地也会起到缓存作用，即使当 Git 服务器无法访问的时候，各个微服务系统仍然可以获取并使用 Config 服务器中的缓存内容。

7.3 使用配置服务

在微服务系统中，各个微服务都是 Config 服务器的客户端，也就可以通过它所提供的各种 HTTP 端点获取所需的配置信息。本节将从客户端角度出发，关注于如何在服务中集成 Config 服务器以及使用位于中央仓库中的配置信息的方法。

仅供非商业用途或交流学习使用

7.3.1 访问配置项

要想通过 Config 服务器获取配置信息，首先需要初始化客户端，然后通过注解将配置信息注入到业务代码中。

1. 初始化客户端

初始化客户端的第一步是引入 Spring Cloud Config 的客户端组件 spring-cloud-config-client，如下所示。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-client</artifactId>
</dependency>
```

然后需要在配置文件 application.yml 中配置 Config 服务器的访问地址，如下所示。

```
spring:
  application:
    name: productservice
  profiles:
    active:
      dev

  cloud:
    config:
       enabled: true
       uri: http://localhost:8888
```

以上配置信息中有几个地方值得注意。首先是 Spring Boot 应用程序的名称“productservice”，该名称必须与 Config 服务器上对应配置文件所在的目录名称保持一致，我们在 7.2.2 节的图 7-4 中已经创建了该目录名称，如果两者无法对应则访问配置信息会发生失败。其次，profile 值为 dev，意味着将使用 dev 环境的配置信息，也就是会获取 Config 服务器上 productservice-dev.yml 配置文件中的内容。最后，需要配置 Config 服务器所在的地址，也就是上例中的 uri，具体为 http://localhost:8888。

以上做法虽说可行，但也存在一个明显的问题，即在配置文件中静态指定了运行时环境，通过 maven 构建出来的 jar 包已经固定为只能用于 dev 环境，那么不同的环境就需要构建不同的 jar 包。更好的做法是只需构建一次 jar 包，然后在运行时改变目标环境和配置信息。Spring Boot 考虑到了这方面的需求，在 java-jar 命令中集成了很多有用的参数用于改变默认的运行

仅供非商业用途或交流学习使用

时行为。这些参数基本上都与 application.yml 文件中的配置项一一对应。例如，想要将服务端口改为 8081，可以使用 jar -jar --server.port=8081 命令来启动某一个 jar 包。针对上文中的案例，可以使用如下命令来设置 Config 服务器的地址为 http://localhost:8889 以及运行环境为 prod 环境。

```
java -jar productservice-0.0.1-SNAPSHOT.jar \
     --spring.cloud.config.uri=http://localhost:8889 \
     --spring.profiles.active=prod
```

2. 通过@Value 注解注入配置信息

初始化客户端的效果相当于在客户端自动集成了访问 Config 服务器中 HTTP 端点的功能。而访问 Config 服务器的过程对于客户端而言是透明的，即客户端不需要考虑如何从远程服务器获取配置信息，而只需要考虑如何在 Spring Boot 应用程序中使用这些配置信息。为此，可以使用 Spring 框架中提供的@Value 注解达到这一目的。

@Value 注解对于 Spring Boot 而言并不是新事物，它位于 org.springframework.beans.factory.annotation 中，可以在初始化时方便地对 Spring 中各种 Bean 的参数变量进行赋值。@Value 注解需要一个 String 类型的值，并使用“${systemProperties.myProp}”形式引用配置文件中的配置信息。

回到 Product 服务中，假设在中央仓库中的 productservice.yml 文件中存在一个自定义配置项“productservice.customproperty”，通常的做法是创建一个 ServiceConfig 类来专门处理配置信息。ServiceConfig 类的代码如下所示，通过@Value 注解可以加载配置文件中的配置信息，其背后复杂的远程 HTTP 端点的请求、配置参数的实例化等过程都由@Value 注解自动完成。

```
@Component
public class ServiceConfig {

    @Value("$(productservice.customproperty)")
    private String customProperty;

    public String getCustomProperty() {
        return customProperty;
    }
}
```

一旦我们能够获取配置文件中的配置信息，就可以根据这些配置信息来完成业务流程。

3. 整合数据库访问功能

在日常的开发过程中，配置文件的常见用途是存储各种第三方工具的元数据，最典型的就

仅供非商业用途或交流学习使用

是管理数据库连接配置。本节将演示通过 Spring Cloud Config 提供的数据库访问数据源（Data Source）的配置过程。

下面以 Product 服务为例来演示数据库访问功能，数据库的集成需要在 Product 服务中引入 jpa 和 mysql 相关的依赖，如下所示。

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
   <groupId>mysql</groupId>
   <artifactId>mysql-connector-java</artifactId>
</dependency>
```

首先定义数据库模式并初始化数据，相关 SQL 脚本如下所示。

```
DROP TABLE IF EXISTS 'product';
CREATE TABLE 'product' (
    'id' bigint(20) NOT NULL AUTO_INCREMENT,
    'description' varchar(255) DEFAULT NULL,
    'price' float DEFAULT NULL,
    'product_code' varchar(255) DEFAULT NULL,
    'product_name' varchar(255) DEFAULT NULL,
    PRIMARY KEY ('id')
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;

INSERT INTO 'product' VALUES ('1', 'New Book For Microservie By Tianyalan',
    '100', 'code1', 'Microservie Practices');
```

然后定义 Product 的实体类，使用了 JPA 相关的@Entity、@Id 和@GeneratedValue 注解，如下所示。我们在数据库模式设计与实体类字段定义上使用了大量默认的命名方法。

```
@Entity
public class Product {
    @Id
    @GeneratedValue
    private Long id;

    private String productCode;
    private String productName;
    private String description;
    private Float price;
```

仅供非商业用途或交流学习使用


```
    省略 get/set
}
```

为了演示 Config 服务器的功能，这里简单设计一个 ProductRepository 并提供 findByProductCode()方法。以 ProductRepository 为基础，ProductService 和 ProductController 的结构也非常简单，各个类的代码如下所示。

```
public interface ProductRepository
    extends JpaRepository<Product, Long> {

    Product findByProductCode(String productCode);
}

@Service
public class ProductService {

    @Autowired
    private ProductRepository productRepository;

    public Product getProductByCode(String productCode) {

        return productRepository.findByProductCode(productCode);
    }
}

@RestController
@RequestMapping(value="v1/products")
public class ProductController {

    @Autowired
    private ProductService productService;

    @RequestMapping(value = "/{productCode}", method =
        RequestMethod.GET)
    public Product getProduct(@PathVariable String productCode) {

        Product product =
            productService.getProductByCode(productCode);
        return product;
    }
}
```

通过 Postman 访问 http://localhost:8081/v1/products/001 端点，可以获取如图 7-8 所示的效果图，表明数据库访问操作成功完成，位于 Config 服务器中的配置信息已经生效。

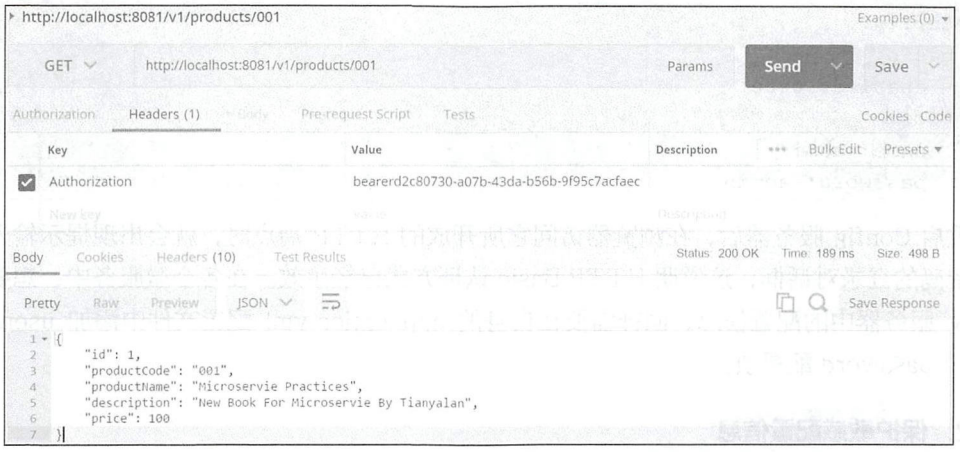


图 7-8　Product 服务集成数据库访问效果示意图

我们可以看到在整合数据库访问功能的整个过程中，开发人员几乎不需要做任何事情就能实现数据库访问，基于 Spring Cloud Config 的配置中心解决方案提供了配置信息统一管理的入口。

7.3.2　配置数据安全性

本节将介绍确保配置数据安全性的两个维度，一个维度是从 Config 服务器的角度出发确保位于服务器上的配置信息不被随意访问，另一个维度则是从数据传输的角度出发确保敏感配置信息得到应有的保护。

1. Config 服务器用户访问认证

通过前面的示例，可以看到 Config 服务器在默认情况下是允许随意访问的，不需要提供任何认证信息就能获取 HTTP 端口所开放出来的各项数据，这其中也包含存储在中央仓库中的配置信息。显然，从安全性角度来讲，这可能是一种潜在的风险。

为了实现针对 Config 服务器的用户访问认证，要做的事情就是添加相应的安全配置项。首先需要引入 spring-boot-starter-security 依赖，如下所示。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

然后在 application.yml 配置文件中添加如下配置信息，表明我们将使用 HTTP Basic 的认证方式来实现访问控制。

```
security:
  basic:
    enabled: true
  user:
    name: admin
    password: admin
```

重启 Config 服务器后，在浏览器访问它所开放的 HTTP 端点时，就会出现提示输入用户名和密码的登录对话框，这说明 HTTP Basic 认证方式已经生效。在各个微服务中，想要获取 Config 服务器中的配置信息，同样需要在自身的 application.yml 配置文件中添加 user.name 和 user.password 配置项。

2. 保护敏感配置信息

回想 7.3.1 节中提到的整合数据库访问功能，我们在配置文件中添加了访问 MySQL 数据库相关的数据源信息，其中的密码“default_pwd”直接以明文方式进行存储。从安全性角度讲，这种做法是存在缺陷的，因为密码是一种敏感配置信息，需要进行保护以免信息泄露。对于敏感配置信息的保护，常见的解决方案是数据加密，具体有对称与非对称加密两种实现方式。下面以对称加密为例，介绍在 Spring Cloud Config 中如何保护敏感配置信息。

Config 服务器的加解密机制依赖于 JCE（Java Cryptography Extension）组件，该组件是一组工具包，提供用于加密、密钥生成和协商等方面的支持。JCE 组件无法通过引入 maven 依赖的方式加载到应用中，需要手工从 Oracle 官网上下载对应版本的 JCE 并替换本地 JDK 环境中/jre/lib/security 下的 local_policy.jar 和 US_export_policy.jar 这两个 jar 包。

Config 服务器提供了专门用于加密和解密的 HTTP 端点，分别是 encrypt 和 decrypt 端点，通过这两个端点可以完成信息的加解密操作。同时，encrypt 端点还会给出当前加密环境的状态，通过访问 http://localhost:8888/encrypt/status 可以获取这种状态信息。在初始状态下，访问该端点的结果如下所示，告诉用户需要设置所需的秘钥信息。

```
{
    "description":"No key was installed for encryption service",
    "status":"NO_KEY"
}
```

对于对称加密而言，公私钥是共用的，只需要在配置文件中指定一个秘钥信息即可，配置内容如下所示。再次访问状态端点，会发现加密环境已经处于正常状态。

```
encrypt:
    key: configsymmetrickey
```

现在我们已经可以通过 encrypt 和 decrypt 端点完成用户信息的加解密，图 7-9 展示了这

一过程，将明文数据 default_pwd 通过 POST 请求发送到 http://localhost:8888/encrypt 端点，就可以获取该数据对应的密文数据。

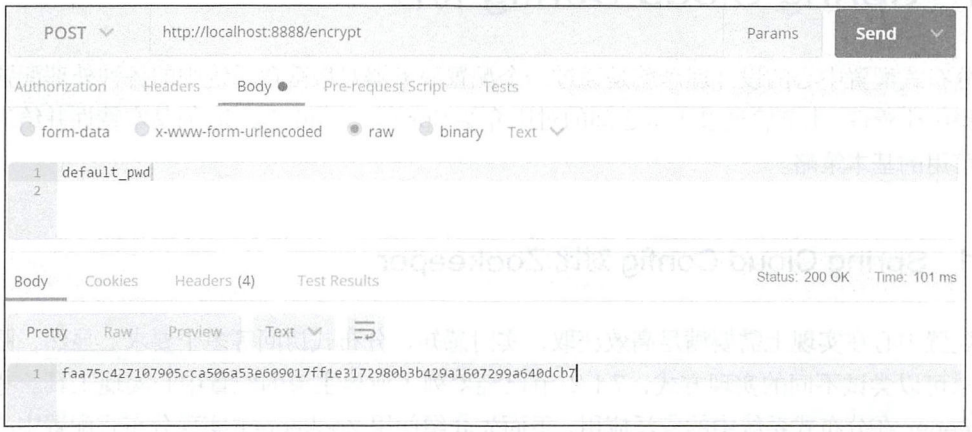


图 7-9　使用 encrypt 端点进行信息加密效果示意图

获取 default_pwd 的密文数据之后，可以使用如下语法修改配置文件 productservice.yml 中的原有密码信息，其中，“{cipher}”前缀用来告诉 Config 服务器这是一个加密之后的数据。

```
spring:
  datasource:
    password:
{cipher}faa75c427107905cca506a53e609017ff1e3172980b3b429a1607299a640dcb7
```

重启 Config 服务器，再次访问 http://localhost:8888/productservice/default 会得到新的配置数据。但在返回的 JSON 数据中仍然发现了如下配置项：“spring.datasource.password: default_pwd”，可以看到加密之后的密文已经被 Config 服务器自动解密到明文状态。自动解密数据是 Config 服务器的默认行为，通过如下配置项可以关闭该功能。

```
spring:
   cloud:
     config:
       server:
          encrypt:
             enable: false
```

关闭 Config 服务器的自动解密功能之后，解密操作将由客户端应用程序自行处理。关于微服务架构中的安全性问题，将在第 9 章中进行更深入的讨论。

7.4 Spring Cloud Config 特性

分布式配置中心的设计理念就是通过一个配置服务器对散落在系统中的各种外部配置文件进行集中化管理。本节将通过工具之间的对比介绍 Spring Cloud Config 的基本特性并给出其实现高可用的基本策略。

7.4.1 Spring Cloud Config 对比 Zookeeper

配置中心在实现上需要满足高效获取、实时感知、分布式访问等多个要求。显然，围绕这些要求可以尝试不同的实现方式，7.1.2 节已经罗列了业界主流的配置中心实现工具。考虑到 Zookeeper 在分布式系统中的广泛应用，下面先介绍使用 Zookeeper 实现分布式配置中心的基本原理，然后在实现机制上对 Spring Cloud Config 和 Zookeeper 做简要对比。

1. 分布式协调与 Zookeeper

为了满足分布式配置中心的实现需求，分布式协调（Distributed Coordination）机制是一个切入点，即通过一定的方法确保配置信息在分布式环境中的各个节点中得到实时、一致的管理。这里将基于目前业界主流的分布式协调框架 Zookeeper 介绍配置中心的实现思路。对 Zookeeper 的详细介绍不是本书的重点，读者可以参考相关资料。

Zookeeper 的核心是一个精简的文件系统抽象，提供基于目录节点树方式的数据存储，以及一些额外的抽象操作，如排序、通知和监控等。Zookeeper 物理结构就是一个文件系统，每一个节点被称为 ZNode，代表该节点位于文件系统中的路径，同时也用于存储数据。

Zookeeper 提供了一些核心组件用于实现分布式协调。例如，Zookeeper 对每个节点都可以设置监听器（Watcher），节点监听机制可以用来实现实时感知，即当某一个节点的信息有任何变动时，所有监听该节点的其他节点都可以实时获取通知，从而做出响应。相对配置中心而言，所有服务都是 Zookeeper 的客户端，这些服务通过对包含配置信息的 Zookeeper 节点进行监听就能获取配置信息更新内容。基于 Zookeeper 实现配置中心的示意图如图 7-10 所示，可以看到该机制同时也提供了分布式访问的效果。

使用 Zookeeper 实现分布式配置中心本质上就是对 Zookeeper 节点和监听器的合理利用。

2. Spring Cloud Config 基本架构

从基本架构上讲，Spring Cloud Config 所采用的设计思路与 Zookeeper 完全不同，其基本架构如图 7-11 所示。

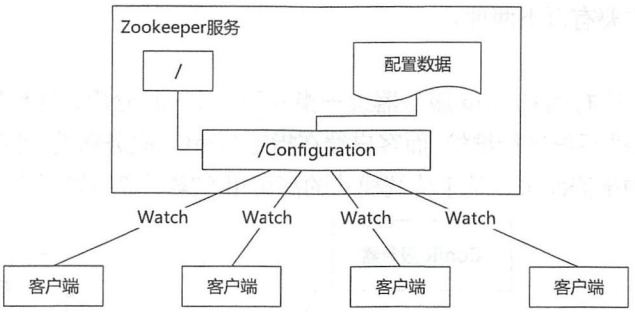


图 7-10　基于 Zookeeper 实现配置中心示例图

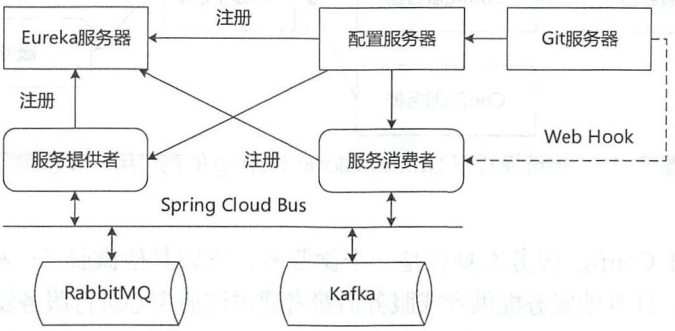


图 7-11　Spring Cloud Config 基本架构

在图 7-11 中，从架构上就可以看出 Spring Cloud Config 与 Zookeeper 的主要区别在于以下几个方面。

- 配置的存储方式不同

Zookeeper 是把配置信息保存在 Zookeeper 自身的文件系统中，而 Spring Cloud Config 也可以将配置信息保存在服务本地，但更多的场景是推荐使用 Git 等版本控制工具来存储配置信息。

- 配置的管理方式不同

Spring Cloud Config 是把配置当成源码来看待并通过 Git 进行管理，Git 的管理界面就是配置的管理界面。而基于 Zookeeper 的实现方案则需要自己开发管理界面。

- 配置变化的通知机制不同

Zookeeper 中的配置变化依赖其自身的事件 Watcher 机制来通知客户端；而 Spring Cloud Config 则是依赖 Git 每次 push 后触发的 webhook 回调并发送事件到 Spring Cloud Bus 消息总线，然后由消息总线通知相关的应用。

7.4.2　Spring Cloud Config 高可用

配置中心作为整个微服务架构运行的基础服务，需要确保其可用性，Spring Cloud Config

实现高可用的方式主要有以下两种。

- 传统模式

在传统模式下，所有的 Config 服务器统一指向同一个 Git 仓库，这样所有的配置内容就通过基于 Git 的共享文件系统进行维护。而客户端在指定 Config 服务器地址时，只需要设置 Config 服务器上层的负载均衡器即可。基于传统模式的高可用方案示意图如图 7-12 所示。

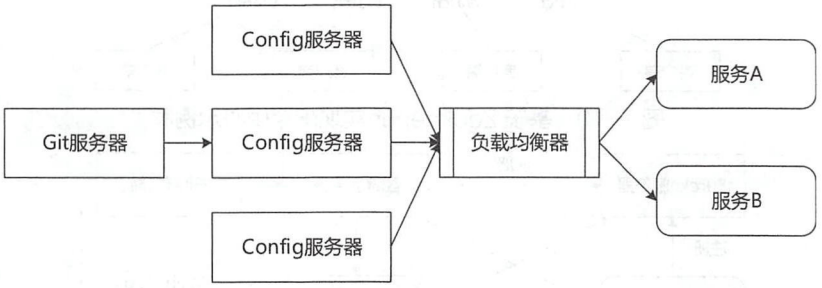


图 7-12　传统模式下 Spring Cloud Config 的高可用方案示意图

- 服务模式

Spring Cloud Config 服务本身也是一个微服务，它跟其他微服务一样，也可以注册到 Eureka 服务器上，让其他服务提供者或服务消费者通过注册中心进行服务发现和获取。显然，这种方式比传统模式更加容易维护，因为服务器端的负载均衡和客户端的配置中心指定功能都可通过基于 Eureka 的服务治理机制得到解决，从而也就间接实现了高可用性。

7.5　本章小结

针对分布式环境下的配置管理场景，本章介绍了 Spring Cloud 中的独立组件 Spring Cloud Config，该组件能够满足日常开发中对配置信息管理的需求并提供了诸多优秀的特性。

本章首先围绕分布式配置中心的解决方案展开讨论，详细介绍了在微服务架构中配置中心的定位和作用、配置信息的分类以及配置中心的实现需求。业界关于配置中心的实现工具有很多，这些工具的背后都具有一定的共性，这些共性是我们理解 Spring Cloud Config 及其他工具的基础。

使用 Spring Cloud Config 涉及服务器端和客户端两方面内容。一方面，作为服务器端我们需要通过 Spring Cloud Config 来构建 Config 服务器以及存放具体配置信息的中央仓库，这里介绍了两种中央仓库的实现方案，一种是本地文件系统，一种是 Git。另一方面，作为客户端我们在通过 Spring Cloud Config 获取配置信息的同时，也需要考虑安全性问题，本章对这些方面都做了详细展开。

Spring Cloud Config 在实现机制上有其特殊性，本章最后通过与 Zookeeper 进行对比描述了 Spring Cloud Config 的这些特性，并介绍了 Spring Cloud Config 实现高可用的解决方案。

第 8 章

Spring Cloud Stream 与事件驱动

本章将介绍事件驱动架构（Event-Driven Architecture，EDA）及其在微服务架构中的应用。事件驱动架构定义了一个设计和实现应用系统的架构风格，在这个架构风格里事件可传输于松散耦合的服务和服务之间。

在微服务设计和开发过程中存在这样的需求，当系统中的某个服务因为用户操作或内部行为发布一个事件，该服务知道这个事件在将来的某个时间点会被其他某个服务所消费，但是并不知道这个服务具体是谁，也不关心什么时候被消费。同样，消费该事件的服务也不一定需要知道该事件是由哪个服务发布的。满足以上场景的系统代表着一种松耦合的架构，也就是事件驱动架构。

事件驱动架构的基本组成如图 8-1 所示，包括事件发布（Publish）、订阅（Subscribe）和消费（Consume）等基本过程。微服务系统中的某一个服务发布事件时，该服务可以广播一个或多个事件到事件中心（Event Center），而每一个对该事件感兴趣的服务都可以订阅这种事件。每当事件被传播时，系统将负责自动调用那些已经订阅了该事件的服务，服务中的事件处理程序将被触发。每个事件消费者都可以有一套独立的事件处理程序，事件发布者并不关心它所发布的事件被如何消费。

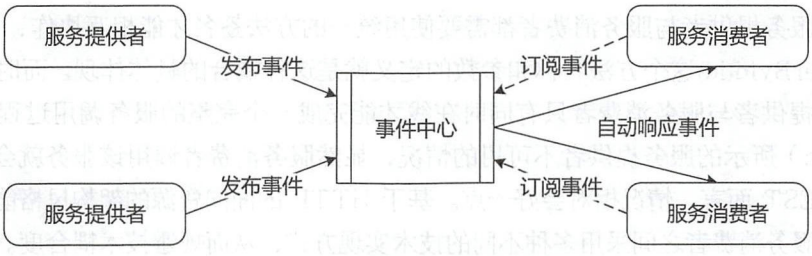


图 8-1　事件处理系统组成结构图

事件驱动架构代表的是一种架构设计风格，其实现方法和工具有很多。Spring Cloud 家族提供了一个专门在微服务系统中实现事件驱动架构的工具 Spring Cloud Stream。本章将引入 Spring Cloud Stream 并结合案例介绍如何进行事件建模。

8.1 事件驱动架构与模型

事件驱动代表的是一种架构模式，而 Spring Cloud Stream 只是这种架构模式的一个实现工具而已。在介绍具体工具之前，我们有必要先对工具背后的架构模式做深入分析，这就是本节的主要内容。

8.1.1 基本事件驱动架构与实现机制

事件驱动架构的出现有其必然性，用户希望使用它来实现一种更为灵活的集成方式。在微服务架构中，引入事件驱动的设计思想也是基于这一目标。

1. 事件驱动架构的需求

在微服务系统中引入事件驱动架构的需求主要来自两个方面，一方面是如何降低各个服务之间的耦合度，另一方面则考虑如何提升系统的扩展性。

（1）耦合度

从软件设计的耦合度上讲，无论是 RPC 还是 REST 都存在一定的耦合度问题。就 RPC 而言，存在 3 种耦合度，即技术耦合、空间耦合和时间耦合。技术耦合度表现在服务提供者与服务消费者之间需要使用同一种技术实现方式，图 8-2（a）中服务提供者与服务消费者都使用 RMI（Remote Method Invocation）作为通信的基本技术，而 RMI 是 Java 领域特有的技术，这就意味着其他服务消费者想要使用该服务也只能采用 Java 作为它的基本开发语言。空间耦合度指的是服务提供者与服务消费者都需要使用统一的方法签名才能相互协作，图 8-2（b）中的 getUserById(id)这个方法名称和参数的定义就是这种耦合的具体体现。而时间耦合度则表现在服务提供者与服务消费者只有同时在线才能完成一个完整的服务调用过程，如果出现了图 8-2（c）所示的服务提供者不可用的情况，显然服务消费者调用该服务就会失败。

对于 REST 而言，情况相对会好一点。基于 HTTP 的面向资源的架构风格能够支持在服务提供者与服务消费者之间采用多种不同的技术实现方式，从而规避技术耦合度。对于空间耦合，也可以采用 HATEOAS（Hypermedia as the Engine of Application State，基于超媒体的应用状态引擎）在一定程度上缓解这种耦合度。但在时间耦合度上，REST 面临与 RPC 同样的场景和问题。

（2）扩展性

在案例系统 OrderSystem 中存在订单处理相关的业务场景，常见的订单处理涉及商品、用户账户和订单自身信息维护等功能，前面章节中已经介绍了 Product 服务、Account 服务和

Order 服务这 3 个微服务。显然，这 3 个服务之间需要进行服务之间的调用和协调从而完成业务闭环（见图 8-3）。如果在不久的将来，OrderSystem 中需要引入其他服务才能形成完整的业务流程，那么图 8-3 所示的交互模式就需要进行相应调整。

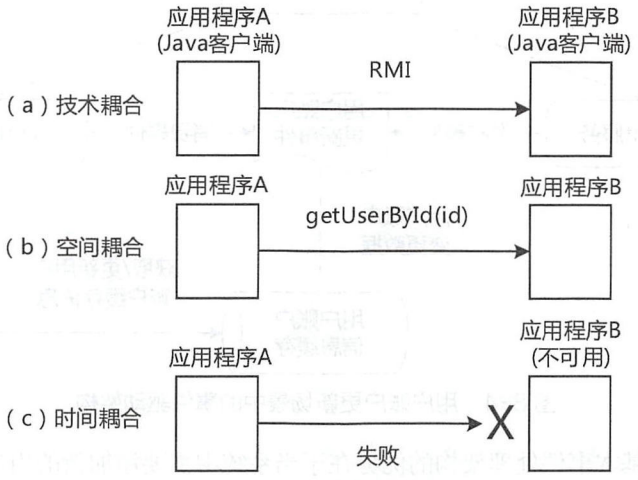


图 8-2　RPC 框架的主要问题

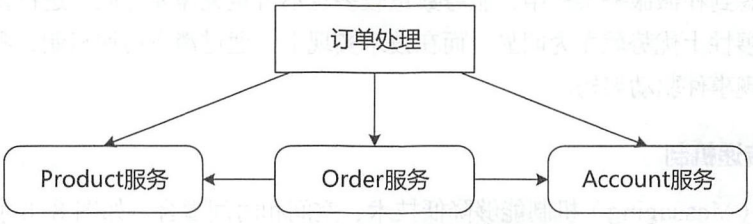


图 8-3　订单处理业务服务交互模式

一般而言，一个用户的账户信息变动并不会太频繁，所以很多时候我们会想到通过缓存系统来存放用户的账户信息，并在订单处理过程中直接从缓存中获取下单用户的用户账户。在这样的设计和实现方式下，试想一旦某个用户账户信息发生变化，应该如何正确和高效地应对这一场景？

考虑到系统扩展性，显然图 8-3 所示的通过访问 Account 服务实时获取用户账户信息的服务交互模式并不是一个好的选择，因为用户账户更新的时机无法事先预知。事件驱动架构为用户提供了一种更好的实现方案。当用户账户信息变更时，Account 服务可以发送一个事件，该事件表明了某个用户账户信息已经发生变化，并将触发所有对该事件感兴趣的微服务，这些微服务就相当于这个事件的订阅者和消费者。通过这种方式，某个特定服务就可以获取用户账户变更事件从而正确且高效地更新缓存信息。原则上我们应该专门开发一个独立的服务来处理事

件，但为了简便起见，本章还是使用 Order 服务作为用户账户信息变更事件的消费者，整个场景的示意图如图 8-4 所示。

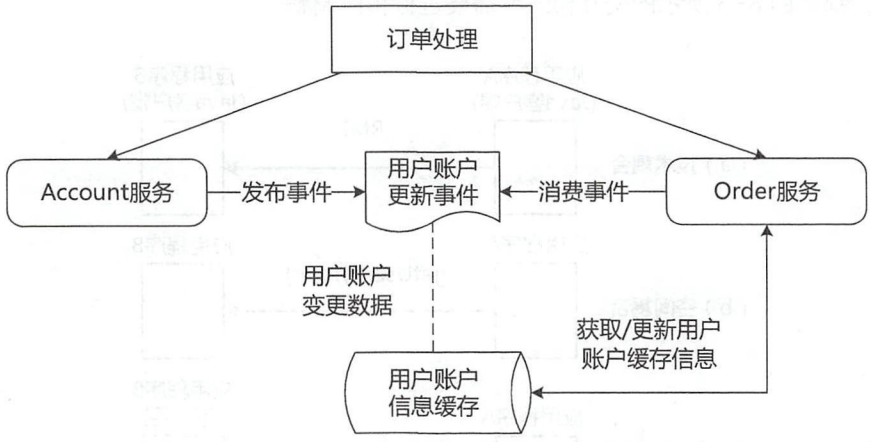


图 8-4　用户账户更新场景中的事件驱动架构

在图 8-4 中，基本事件处理架构的优势在于当系统中需要添加新的用户账户变更事件处理逻辑来完成整个流程时，我们只需要对该事件添加一个订阅者即可，不需要对订单处理系统做任何修改。考虑到在微服务架构中，服务数量较多且不可避免需要对服务进行重构，事件处理架构在系统扩展性上优势就尤为明显。而在技术实现上，通过消息传递机制，我们不必花费太大代价就能实现事件驱动架构。

2. 消息传递机制

消息传递（Messaging）机制能够降低技术、空间和时间耦合。如图 8-5 所示，消息传递机制在消息发送方和消息接收方之间添加了存储转发（Store and Forward）功能。存储转发是

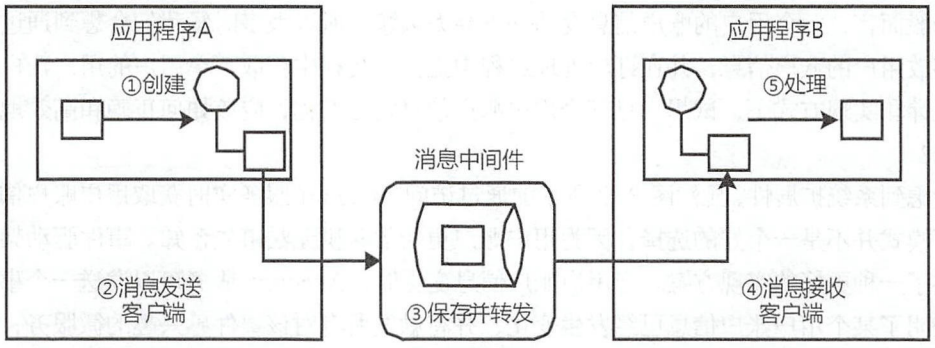


图 8-5　消息传递机制

计算机网络领域应用最为广泛的技术之一，其基本思想是将数据先缓存起来，再根据其目的地

址将该数据发送出去。显然，有了存储转发机制之后，消息发送方和消息接收方之间并不需要相互认识，也不需要同时在线，更加不需要采用同样的实现技术。紧耦合的单阶段方法调用就转变成松耦合的两阶段过程，技术、空间和时间上的约束通过中间层得到显著缓解，这个中间层就是消息传递系统（Messaging System）。

在消息传递系统中，消息的生产者负责产生消息，一般由业务系统充当生产者；消息的消费者负责消费消息，一般是后台系统负责异步消费。生产者行为模式单一，而消费者根据消费方式的不同有一些特定的分类，常见的有推送型消费者（Push Consumer）和拉取型消费者（Pull Consumer），推送指的是应用系统向消费者对象注册一个 Listener 接口并通过回调 Listener 接口实现消息消费，而在拉取方式下应用系统通常主动调用消费者的拉消息方法消费消息，主动权由应用系统控制。

消息传递有两种基本模型，即发布-订阅（Pub-Sub）模型和点对点（Point to Point）模型。发布-订阅模型支持生产者、消费者之间的一对多关系，是典型的推送消费者实现机制；而点对点模型中有且仅有一个消费者，通过基于间隔性拉取的轮询（Polling）方式进行消息消费。当生产者和消费者数量较多时，也可以引入组（Group）的概念，Producer Group 和 Consumer Group 分别代表一类生产者和消费者的集合名称，使用统一逻辑发送和接收消息。

上述概念构成了消息传递系统最基本的模型，围绕这个模型业界有一些实现规范和工具，代表性的规范有 JMS 和 AMQP 以及它们的实现框架 ActiveMQ 和 RabbitMQ 等，而 Kafka、RocketMQ 等工具并不遵循特定的规范但也提供了消息传递的设计和实现方案。因为 Spring Cloud Stream 基于消息中间件，所以在本章中，我们也会对部分消息中间件的实现方案和使用方法做相关介绍。

8.1.2 事件驱动与领域模型

1. 事件订阅者模型

在本章开篇，我们给出了图 8-1 所示的事件驱动架构基本模型，考虑到现实场景中会存在不同类型的事件生产者和消费者，可以在该模型上进行扩展，得到如图 8-6 所示的完整版事件驱动模型。

对于事件生命周期而言，一般存在产生、分发、存储和使用这 4 个阶段。下面结合事件生命周期，对图 8-6 中 3 种不同类型的事件订阅者展开讨论。

（1）简单订阅者

简单订阅者直接处理事件，表现为一个独立的事件处理程序，对应于事件的使用阶段。

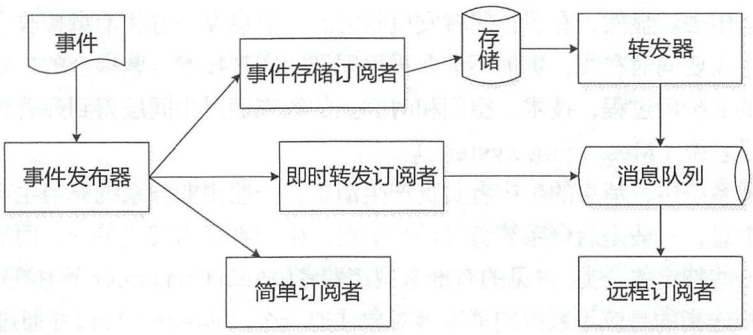


图 8-6　事件驱动模型

（2）即时转发订阅者

即时转发订阅者对应于事件的分发和使用阶段，一方面可以具备简单订阅者的功能，另一方面也可以把事件转发给其他订阅者。通常，把事件转发到消息队列是一个好的方法，很多消息传递系统都具备强大的一对一和一对多转发功能，可以满足远程订阅者处理事件的需求。

（3）事件存储订阅者

事件存储订阅者在处理事件的同时对事件进行持久化。存储的事件可以作为一种历史记录，也可以通过专门的事件转发器转发到消息队列，对应于事件的存储和使用阶段。

2. 领域事件

在《微服务设计原理与架构》和本书第一章中都提到，可以使用面向领域驱动思想来明确服务之间的边界。在领域驱动设计中存在一个专门的用户处理事件的组件，即领域事件（Domain Event）。领域事件的基本结构通常包括唯一标识、产生时间、事件来源、事件类型以及各种业务数据，并且具有不可变性。

领域事件既可以由本地界限上下文消费，也可以由远程的界限上下文消费。远程界限上下文发布领域事件需要考虑消息的最终一致性、同步和异步以及领域事件存储等问题。尤其针对远程交互本身存在的网络稳定性等各种不可控因素，一般都会对事件进行存储以便在发生问题时进行跟踪和重试。支持不同事件类型，支持领域事件和存储事件之间的转换，检索由领域模型产生的所有结果的历史记录，使用事件存储中的数据进行业务预测和分析是常见的事件存储需求。

事件可以通过领域事件发布者（DomainEventPublisher）进行集中式的存储，也可以分别保存在各个领域事件订阅者（DomainEventSubscriber）中。图 8-7 就是一个包含事件存储的发布订阅时序图，通过领域事件发布者构建 EventStore（事件存储器）进行事件存储。

仅供非商业用途或交流学习使用

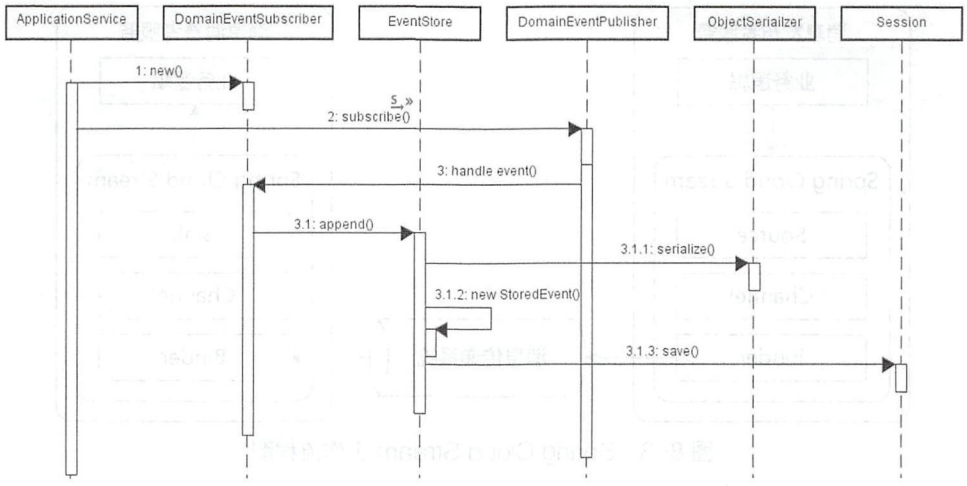


图 8-7　包含事件存储的发布订阅时序图

8.2　引入 Spring Cloud Stream

关于事件驱动架构的实现有两种解决方案：一种是直接使用诸如 RabbitMQ、Kafka 等消息中间件来实现消息传递，这种解决方案需要开发人员掌握各种工具的特性并采用不同的方式进行系统集成；另一种是提供一个整体的平台型解决方案，从而屏蔽各个消息中间件在技术实现上的差异。显然，我们倾向于后者，Spring Cloud Stream 正是这样一种平台型解决方案。本节将首先介绍 Spring Cloud Stream 的基本架构，并给出它与目前主流的各种消息中间件之间的整合机制。

8.2.1　Spring Cloud Stream 基本架构

Spring Cloud Stream 对整个消息发布和消费过程做了高度抽象，并提供了一系列核心组件。本节先介绍通过 Spring Cloud Stream 构建事件驱动架构的基本工作流程，然后给出对各个核心组件的详细描述。

1. Spring Cloud Stream 工作流程

Spring Cloud Stream 中有 3 种角色，即消息的发布者、消费者以及消息传递系统本身，以消息传递系统为中心，整个工作流程表现为一种对称结构，如图 8-8 所示。

仅供非商业用途或交流学习使用

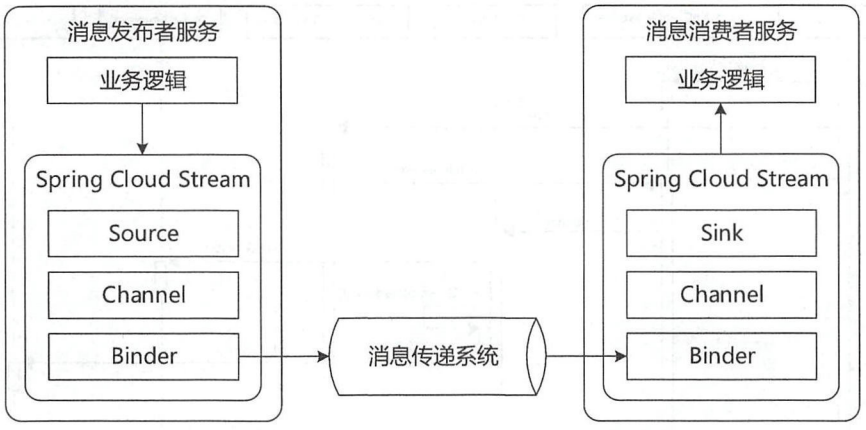


图 8-8　Spring Cloud Stream 工作流程图

在图 8-8 中，消息发布者根据业务需要产生消息发送的需求，而 Spring Cloud Stream 中的 Source 组件是真正生成消息的组件，然后消息通过 Channel 传送到 Binder，这里的 Binder 是一个抽象组件，通过 Binder 可以与特定的消息传递系统进行通信。在 Spring Cloud Stream 中，目前已经内置集成的消息传递系统包括 RabbitMQ 和 Kafka，在 8.2.3 节中会对这两个消息传递系统做简要介绍。

消息消费者同样可以通过 Binder 从消息传递系统中获取消息，消息通过 Channel 将流转到 Sink 组件。这里的 Sink 组件是服务级别的，即每个微服务可能会实现不同的 Sink 组件，分别对消息进行业务上的处理。

2. Spring Cloud Stream 核心组件

从图 8-8 中可以看出 Spring Cloud Stream 具备 4 个核心组件，分别是 Binder、Channel、Source 和 Sink。其中，Binder 和 Channel 成对出现，而 Source 和 Sink 分别面向消息的发布者和消费者。

（1）Binder

Binder 是 Spring Cloud Stream 的一个重要抽象概念，是服务与消息传递系统之间的黏合剂。目前 Spring Cloud Stream 实现了面向 Kafka 和 RabbitMQ 这两种消息中间件的 Binder。通过 Binder，可以很方便地连接消息中间件，可以动态地改变消息的目标地址、发送方式，而不需要了解其背后的各种消息中间件在实现上的差异。

（2）Channel

Channel 即通道，是对队列（Queue）的一种抽象。我们知道在消息传递系统中，队列的作用就是实现存储转发的媒介，消息发布者所生成的消息都将保存在队列中并由消息消费者进行消费。通道的名称对应的就是队列的名称，但是作为一种抽象和封装，各个消息传递系统所

仅供非商业用途或交流学习使用

特有的队列概念并不会直接暴露在业务代码中，而是通过通道来对队列进行配置。

（3）Source 和 Sink

在介绍 Source 和 Sink 之前，需要明确两个在消息传递系统和企业服务总线领域中经常碰到的概念，即输入（Inbound）和输出（Outbound）。这里输入/输出的参照对象是 Spring Cloud Stream 自身，即从 Spring Cloud Stream 发布消息就是输出，而通过 Spring Cloud Stream 接收消息就是输入。因此，Source 组件用于面向单个输出通道的应用，而 Sink 组件则用于有单个输入通道的应用。

在 Spring Cloud Stream 中，Source 组件就是使用一个 POJO（Plain Old Java Object）对象来作为需要发布的消息，将该对象进行序列化（默认的序列化方式是 JSON）后发布到通道中。而 Sink 组件用于监听通道并等待消息的到来，一旦有可用消息，Sink 就会该消息反序列化为一个 POJO 对象并用于处理业务逻辑。

8.2.2 Spring Cloud Stream 与 Spring Integration

在了解了 Spring Cloud Stream 的基本流程和核心组件之后，我们来看一下该框架背后的实现机制。事实上，Spring Cloud Stream 是基于 Spring Integration 实现了消息发布和消费机制并提供了一层封装，很多关于消息发布和消费的概念及实现方法本质上都是依赖于 Spring Integration。本节将简要介绍 Spring Integration 框架，以方便读者在使用 Spring Cloud Stream 时能理解其背后的实现原理。

1. Spring Integration 简介

Spring Integration 是 Spring 家族中的一员，作为轻量级、松耦合集成框架，与现有 Spring 应用程序能够完美融合。从定位上讲，Spring Integration 是一种企业服务总线，能支持并扩展主流系统集成模式，并提供众多基础性系统交互端点技术。

Spring Integration 把通道抽象成两种基本的表现形式，即支持轮询的 PollableChannel 和实现发布-订阅模式的 SubscribableChannel，这两个通道都继承自具有消息发送功能的 MessageChannel，通道相关的定义如下所示。

```
public interface MessageChannel {

    boolean send(Message message);

    boolean send(Message message, long timeout);
}

public interface PollableChannel extends MessageChannel {
```

仅供非商业用途或交流学习使用

```
    Message<?> receive();

    Message<?> receive(long timeout);
}

public interface SubscribableChannel extends MessageChannel {

    boolean subscribe(MessageHandler handler);

    boolean unsubscribe(MessageHandler handler);
}
```

我们注意到对于 PollableChannel 而言才有 receive()的概念，代表这是通过轮询操作主动获取消息的过程，而 SubscribableChannel 则是通过注册回调函数 MessageHandler 来实现事件响应。Spring Integration 同时实现了其他几种通道，包括阻塞式队列的 RendezvousChannel，该通道与带缓存的 QueueChannel 都属于点对点通道，只有等前一个消息被消费之后才能发送下一个消息。PriorityChannel 即优先级队列，而 DirectChannel 是 Spring Integration 的默认通道，该通道的消息发送和接收过程处于同一线程中。另外还有 ExecutorChannel，使用基于多线程的 TaskExecutor 来异步消费通道中的消息。

Spring Integration 提供了 PayloadTypeRouter 和 HeaderValueRouter 用于实现基于内容的消息路由(Message Routing)，当然对于任何需要定制化的路由场景，我们都可以像在 Spring 中注入一个普通的 Javabean 一样把路由逻辑嵌入到路由器中。

同时，Spring Integration 内置一批基本数据结构转换器（Transformer），包括 object-to-string-transformer、object-to-map-transformer、json-to-object-transformer 等。Header Enricher、Payload Enricher、Header Filter 等基于内容的扩充和过滤工具在 Spring Integration 中也都有提供实现。

Spring Integration 还提供了消息网关（Gateway）和服务激励器（Service Activator）等专用的端点类型，本书无意对这些内容做过多阐述，关于 Spring Integration 更多信息请参考其官方网站，相关的企业应用集成模式也可以参考 Gregor Hohpe 和 Bobby Woolf 的经典著作《企业集成模式》。

2. Spring Cloud Stream 中的 Spring Integration

结合 Spring Integration 中的相关概念，就不难理解 Spring Cloud Stream 中关于 Source 和 Sink 的定义了。Source 和 Sink 都是接口，Source 接口的定义如下，通过 MessageChannel 来发送消息，而@Output 注解定义了一个输出通道，发布的消息通过该通道离开应用。

```
import org.springframework.cloud.stream.annotation.Output;
import org.springframework.messaging.MessageChannel;
```

仅供非商业用途或交流学习使用

```
public interface Source {

    String OUTPUT = "output";

    @Output(Source.OUTPUT)
    MessageChannel output();
}
```

类似的，Sink接口定义如下，通过Spring Integration中的SubscribableChannel来实现消息接收，而@Input注解定义了一个输入通道，应用通过该输入通道接收来自外部的消息。

```
import org.springframework.cloud.stream.annotation.Input;
import org.springframework.messaging.SubscribableChannel;

public interface Sink{

    String INPUT = "input";

    @Input(Source.INPUT)
    SubscribableChannel input();
}
```

注意，@Input和@Output注解可以使用通道名称作为参数，如果没有名称，会使用带注解的方法名字作为参数，也就是默认情况下分别使用“Input”和“Output”作为通道名称。从这个角度讲，一个Spring Cloud Stream应用程序可以存在任意数量的Input和Output通道，只需要对这些通道通过@Input和@Output注解进行定义即可。例如，在以下接口中定义了MyChannel接口，并声明了一个Input通道和两个Output通道，说明该服务会向外部的两个通道发送消息并从外部的一个通道中获取消息。

```
public interface MyChannel{

    @Input
    SubscribableChannel input1();

    @Output
    MessageChannel output1();

    @Output
    MessageChannel output2();
}
```

上述接口定义中使用了Spring Integration中的SubscribableChannel和MessageChannel接口，Spring Cloud Stream对Spring Integration提供了原生支持，可以使用Spring

仅供非商业用途或交流学习使用

Integration 提供的 API 直接操作消息发布和接收的过程。但在多数场景下，我们不需要依赖这种方式就能完成常见的开发需求，所以本书不对这种方式做具体展开。

8.2.3 Spring Cloud Stream 与消息中间件

Spring Cloud Stream 中的 Binder 组件是服务与消息中间件之间的一层抽象，而各种消息中间件在消息传递机制的设计和实现上存在差异性。在本小节中，我们梳理 Spring Cloud Stream 中的消息传递模型，并给出 Binder 与消息中间件进行整合的过程。

1. Spring Cloud Stream 中的消息传递模型

Spring Cloud Stream 将消息发布和消费抽象成如下 3 个核心概念，并结合目前主流的一些消息中间件对这些概念提供了统一的实现方式。

（1）发布-订阅模型

消息传递有两个基本模型，即点对点模型和发布-订阅模型，其中，点对点模型可以被视为发布-订阅模型在订阅者数量为 1 时的一种特例。在 Spring Cloud Stream 中，统一通过发布-订阅模型完成消息的发布和消费，即当一条消息被投递到消息中间件之后，所有订阅该消息的消费者都可以消费到这条消息。

（2）消费组

在微服务架构中，每个微服务为了实现高可用和负载均衡都会部署多个服务实例。显然，如果采用发布-订阅模式就会导致同一条消息被同一服务的不同实例重复消费。为了解决这个问题，Spring Cloud Stream 中提供了消费者组（Consumer Group）的概念。一旦使用了消费者组，则一条消息只能被同一个组中的某一个服务实例所消费。消费者组的基本结构如图 8-9 所示。

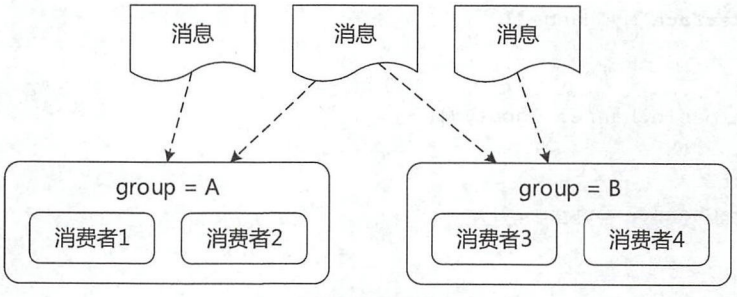


图 8-9 消费者组结构示意图

（3）消息分区

假如想让相同的消息都被同一个微服务实例来处理，但又有多个服务实例组成了负载均衡，

仅供非商业用途或交流学习使用

通过上述的消费者组仍然不能满足要求。针对这一场景，Spring Cloud Stream 引入了消息分区的概念。当生产者将消息数据发送给多个消费者实例时，保证同一消息始终是由同一个消费者实例接收和处理。尽管消息分区的应用场景并没有那么广泛，但如果想要达到类似的效果，Spring Cloud Stream 也为我们提供了一种简单的实现方案，消息分区的基本结构如图 8-10 所示。

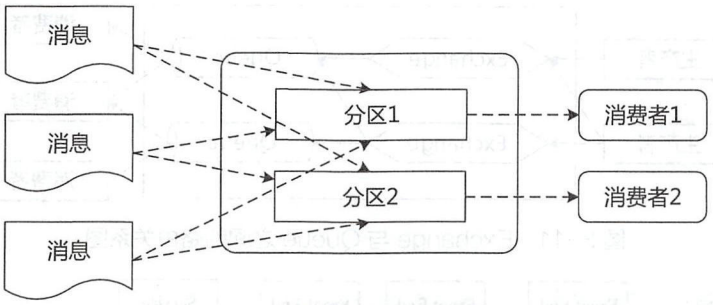


图 8-10　消息分区结构示意图

2. Binder 与消息中间件

Binder 组件作为 Spring Cloud Stream 中的核心组件，本质上是一个中间层，负责与各种消息中间件的交互。目前，Spring Cloud Stream 提供了对 RabbitMQ 和 Kafka 这两个主流消息中间件的集成。在具体介绍使用 Spring Cloud Stream 发布和消费消息之前，先来结合消息传递核心概念给出 Binder 对这两种不同消息中间件的整合方式。

（1）RabbitMQ

在 RabbitMQ 中，没有显式地指明类似 JMS（Java Message Service，Java 消息服务）规范中一对一的点对点模型和一对多的发布-订阅模型，但通过控制交换器（Exchange）与 Queue 之间的路由规则我们可以很容易模拟出存储转发和主题订阅这些典型的消息中间件概念。一个 RabbitMQ 中可能存在多个 Queue，Exchange 如何知道它要把消息发送到哪个 Queue 中去呢？关键就在于通过绑定（Binding）规则设置的路由信息。在与多个 Queue 关联之后，Exchange 中就会存在一个路由表，这个表中存储着每个 Queue 所能存储消息的限制条件。消息的 Header 中有个属性叫路由键（Routing Key），它由消息发送者产生，提供给 Exchange 路由这条消息的标准。Exchange 会检查 Routing Key 并结合路由算法来决定将消息发送到哪个 Queue 中去。图 8-11 就是 Exchange 与 Queue 之间的路由关系图，可以看到一条来自生产者的消息通过 Exchange 中的路由算法可以发送给一个或多个 Queue，从而分别实现点对点和发布-订阅功能。

关于消费组，基于图 8-11 也不难得出实现方案。因为 RabbitMQ 里每个 Queue 是被消费者竞争消费的，所以指定同一个组的消费者消费同一个 Queue 即可。

（2）Kafka

Kafka 分布式架构如图 8-12 所示，从中可以看到 Broker、Producer、Consumer、Push、

仅供非商业用途或交流学习使用

Pull 等消息传递系统常见概念都在 Kafka 中有所体现，生产者使用 Push 模式将消息发布到服务器，而消费者使用 Pull 模式从 Broker 订阅消息。我们注意到在图 8-12 中还使用到了 Zookeeper，其作用在于实现服务器与消费者之间的负载均衡，所以启动 Kafka 之前必须确保 Zookeeper 正在正常运行。

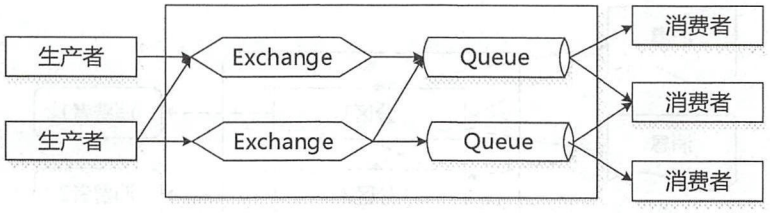


图 8-11　Exchange 与 Queue 之间的路由关系图

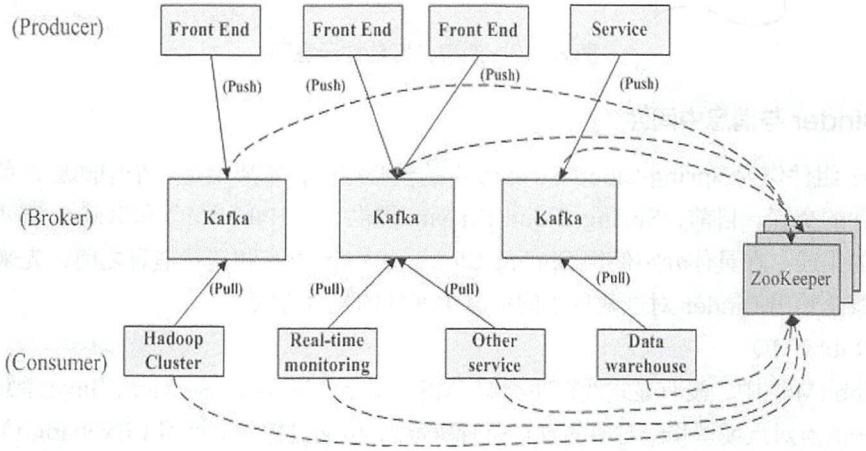


图 8-12　Kafka 分布式架构

同时，Kafka 也实现了消费者组机制（见图 8-13），多个消费者构成组结构，消息只能传输给某个组中的某一个消费者，也就是说消息的消费具有显式的分布式特性，天生就内置了 Spring Cloud Stream 中的消费组概念。

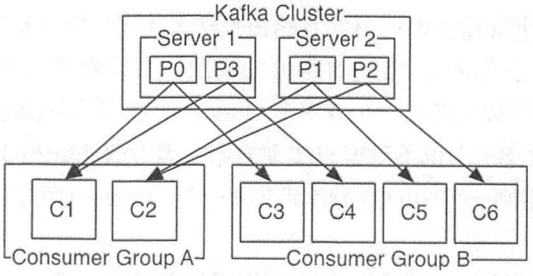


图 8-13　Kafka 消费者分组

仅供非商业用途或交流学习使用

在 Kafka 安装目录中，可以通过 bin/zookeeper-server-start.sh config/zookeeper.properties 命令启动 Kafka，当然也可以在 Zookeeper 已经启动的前提下通过 bin/kafka-server-start.sh config/server.properties 命令达到同样的效果。Kafka 启动成功之后，通过 bin/kafka-topics.sh --create --zookeeper localhost:2181 --topic test 命令可以创建一个名为“test”的主题(Topic)，然后就可以基于该主题发布和消费消息。发布“This is a message”这条消息的命令格式为 bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test This is a message，然后可以使用 bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic test 命令来启动消费者并消费该 Topic 上的消息。这些示例可作为使用 Kafka 的入门参考，关于 Kafka 的更多介绍请参考其官网。

8.3 实现消息发布者

本节和下一节将介绍使用 Spring Cloud Stream 分别实现消息发布者和消息消费者的具体步骤，我们将基于 Kafka 这一特定消息中间件来演示整个过程。

8.3.1 消息发送场景与实现流程

现实中很多场景都可以抽象成事件，如当……发生……时、如果发生……、当……时，通知我等，这些场景都适合发送消息。在案例 OrderSystem 中，Account 服务管理着用户的账户信息，而账户信息显然可以进行变更操作。一旦用户账户信息发生变更，就可以将这个动作建模成一个事件并通过消息将该事件传播出去。

一方面，我们把用户账户信息变更事件命名为 AccountChangedEvent，事件在命名上通常采用过去时态以表示该事件所代表的动作已经发生。另一方面，我们需要设计并实现使用 Spring Cloud Stream 发布消息的各个组件，包括 Source、Channel 和 Binder。图 8-14 展示了围绕 AccountChangedEvent 事件的整个实现流程。

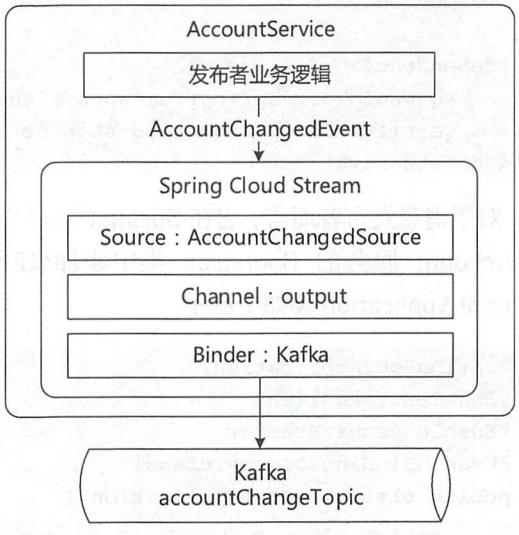


图 8-14 AccountChangedEvent 事件的实现流程

在图 8-14 中，业务逻辑代码根据需

要会触发 AccountChangedEvent 事件，然后该事件将被构建成一个消息并通过 AccountChangedSource 进行发送，AccountChangeSource 封装了对 Spring Cloud Stream 中的 Source 组件的调用。然后 AccountChangeSource 使用默认的名为“output”的 Channel 进行消息发布，而 Binder 组件封装了 Kafka 这一消息中间件。要想把消息发布到 Kafka，我们需要知道 AccountChangedEvent 事件所发送的目的地（Destination），在图 8-14 中这一地址被指定为 accountChangedTopic，accountChangeTopic 是 Kakfa 中的一个特定 Topic，图 8-14 中的 Channel 对该 Topic 做了一层映射。

8.3.2 在服务中添加消息发布者

本节将围绕图 8-14 中 AccountChangedEvent 事件的实现流程展开讨论，并给出各个组件的实现方式和工具。

1. 初始化消息发布环境

无论是消息发布者还是消息消费者，首先都需要引入 spring-cloud-stream 依赖，在以下案例中，将使用 Kafka 作为我们的消息中间件系统，所以也需要引入 spring-cloud-starter-stream-kafka 依赖，如下所示。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-kafka</artifactId>
</dependency>
```

对于消息发布者而言，它在 Spring Cloud Stream 体系中扮演着 Source 的角色，所以需要在 Account 服务的 Bootstrap 类中添加@EnableBinding(Source.class)注解。调整之后的 AccountApplication 类如下所示。

```
@SpringBootApplication
@EnableEurekaClient
@EnableCircuitBreaker
@EnableBinding(Source.class)
public class AccountApplication {

    public static void main(String[] args) {
```

```
        SpringApplication.run(AccountApplication.class, args);
    }
}
```

以上代码中需要展开的是@EnableBinding 注解，该注解的作用就是告诉 Spring Cloud Stream 去绑定该应用程序到消息中间件，实现两者之间的连接。@EnableBinding 注解可以使用一个或者多个接口作为参数，上面的代码中使用了 Source 接口，表示与消息中间件绑定的是一个消息发布者应用程序。接下来就需要通过 Source 接口来具体实现消息的发布。

2. 创建 Event 和 Source

首先我们给出了 AccountChangedEvent 的定义，该事件包括事件类型、事件所对应的操作、事件对应的业务领域对象。AccountChangedEvent 类的定义如下所示。

```
public class AccountChangedEvent{

    //事件类型
    private String type;
    //事件所对应的操作
    private String action;
    //事件对应的领域模型
    private Account account;

    public AccountChangedEvent(String type, String action, Account
        account) {
        super();
        this.type    = type;
        this.action = action;
        this.account = account;
    }
    省略 get()/set()方法
    …
}
```

消息发送的实现类 AccountChangeSource 代码如下，可以看到我们首先构建了 AccountChangedEvent 事件并通过 MessageBuilder 工具类将它转换为消息中间件所能发送的 Message 对象，然后通过 Source 接口的 output()方法发布了该事件，这里的 output()方法背后使用的就是一个具体的 Channel。

```
@Component
public class AccountChangeSource {
    private Source source;
```

```
    private static final Logger logger =
        LoggerFactory.getLogger(AccountChangeSource.class);

    @Autowired
    public AccountChangeSource(Source source){
        this.source = source;
    }

    private void publishAccountChange(String action, Account account){
        logger.debug("Sending Kafka message for Account Id: {}",
            account.getId());

        AccountChangedEvent change =  new AccountChangedEvent(
            AccountChangedEvent.class.getTypeName(),
            action,
            account);

        source.output().send(
            MessageBuilder.withPayload(change).build());
    }

    public void publishAccountAddedEvent(Account account) {
        publishAccountChange("ADD", account);
    }

    public void publishAccountUpdatedEvent(Account account) {
        publishAccountChange("UPDATE", account);
    }

    public void publishAccountDeletedEvent(Account account) {
        publishAccountChange("DELETE", account);
    }
}
```

3. 配置 Binder

为了通过 AccountChangeSource 将消息发送到正确的地址，需要在 application.yml 配置文件中配置 Binder 信息。相关配置项如下所示，可以看到这里的“bindings.output”配置段代表发布消息的 Channel，其目标地址是“accountChangeTopic”。“bindings.kafka”配置段使用 Kafka 作为消息中间件平台，并将其 Zookeeper 地址及 Kafka 自身的地址都指向了本地。

```
spring:
  cloud:
    stream:
      bindings:
        output:
          destination:  accountChangeTopic
          content-type: application/json
        kafka:
          binder:
            zkNodes: localhost
            brokers: localhost
```

4. 集成服务

最后，我们要做的事情就是在业务服务中集成消息发布功能。用户账户信息的变更涉及的业务服务是 AccountService，其代码如下所示，用户账户的增加、修改和删除操作都需要发布用户账户信息变更事件。在 AccountService 中我们并没有构建具体的 AccountChangedEvent 对象，而是把这部分操作放在了 AccountChangeSource 中，其目的就是为了降低各个层次之间的依赖关系，并封装对事件的统一操作。

```
@Service
public class AccountService {
    @Autowired
    private AccountRepository accountRepository;

    @Autowired
    private AccountChangeSource accountChangeSource;

    public Account getAccountById(Long accountId) {
        return accountRepository.findById(accountId);
    }
    public void saveAccount(Account account){
        accountRepository.save(account);

        accountChangeSource.publishAccountAddedEvent(account);
    }
    public void updateAccount(Account account){
        accountRepository.save(account);

        accountChangeSource.publishAccountUpdatedEvent(account);
    }
    public void deleteAccount(Account account){
        accountRepository.delete(account);
```

```
            accountChangeSource.publishAccountDeletedEvent(account);
        }
    }
```

至此，完整的消息发布者实现完毕，下一节将介绍如何实现消息消费者的方法，在行文思路上与本节内容高度一致。

8.4 实现消息消费者

本节顺着上文的思路，同样基于 Kafka 这一特定消息中间件来演示对上文中 AccountChangedEvent 事件进行消费的整个过程。

8.4.1 消息消费场景与实现流程

原则上对用户账户更新有兴趣的服务都可能会消费 AccountChangedEvent 事件。在案例中，Order 服务就是潜在的消费者，因为添加订单操作需要获取用户账户信息。下面回顾一下 OrderService 类中 addOrder()方法的主体逻辑，代码如下所示。

```
public Order addOrder(Long accountId, String productCode) {
    Order order = new Order();
    order.setId(1L);

    Product product = getProduct(productCode);
    if(product == null) {
        return order;
    }

    Account account = getAccount(accountId);
    if(account == null) {
        return order;
    }

    order.setAccountId(accountId);
    order.setItem(product.getProductName());
    order.setCreateTime(new Date());

    orderRepository.save(order);

    return order;
}
```

以上代码中，getAccount(accountId)方法根据 Id 获取用户账户的详细信息，实现上是通过调用远程接口的方式达到这一目的，原有的代码如下所示。

```
@Autowired
AccountClient accountClient;

@HystrixCommand
private Account getAccount(Long accountId) {

    return accountClient.getAccount(accountId);
}
```

我们知道用户账户信息变更是一个低频事件，而每次通过 AccountClient 实现远程调用的成本很高且没有必要。现在我们可以获取用户账户信息更新的消息了，基本思路是，缓存用户账户信息，并通过消息触发缓存更新。图 8-15 展示了采用这一设计思想之后的时序图。

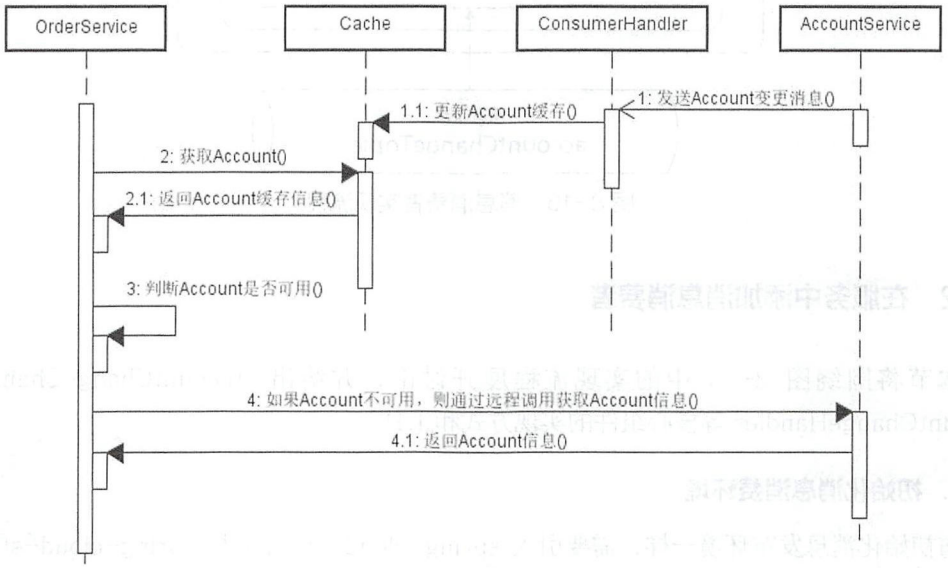


图 8-15　用户账户更新流程时序图

在图 8-15 中，可以看到 AccountService 异步发送的 AccountChangedEvent 事件会被消费该消息的处理器 ConsumerHandler 所消费，然后 ConsumerHandler 将更新后的用户账户信息进行缓存以供 OrderService 使用。ConsumerHandler 是实现消息消费的关键，我们可以独立创建这样的服务，但为了简单起见，这里还是会把 ConsumerHandler 的代码一并放在 OrderService 中。图 8-16 展示了 OrderService 作为消息消费者的实现流程。

在图 8-16 中，AccountChangedEvent 事件通过 Kafka 的 accountChangedTopic 发送到

Spring Cloud Stream 中，Spring Cloud Stream 通过自定义的 AccountChangeChannel 获取消息并交由 AccountChangeHandler 实现具体的消费逻辑，这里的 AccountChangeHandler 就相当于图 8-15 中的 ConsumerHandler。

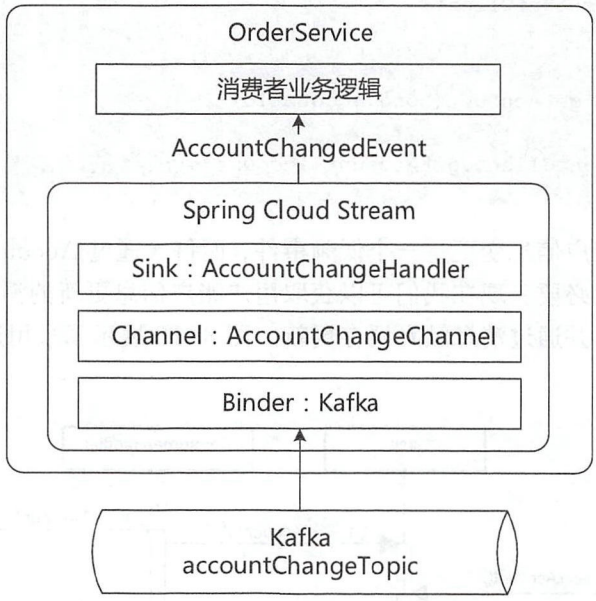


图 8-16　消息消费者实现流程

8.4.2　在服务中添加消息消费者

本节将围绕图 8-16 中的实现流程展开讨论，并给出 AccountChangeChannel 和 AccountChangeHandler 等核心组件的实现方式和工具。

1. 初始化消息消费环境

与初始化消息发布环境一样，需要引入 spring-cloud-stream 和 spring-cloud-starter-stream-kafka 这两个 maven 依赖，并构建 Bootstrap 类。Order 服务中的 Bootstrap 类是 OrderApplication，其代码如下所示。显然，对于消息消费者而言，@EnableBinding 注解所绑定的应该是 Sink 接口。

```
@SpringBootApplication
@EnableEurekaClient
@EnableCircuitBreaker
@EnableBinding(Sink.class)
public class OrderApplication {
```

```
    public static void main(String[] args) {
        SpringApplication.run(OrderApplication.class, args);
    }
}
```

2. 自定义消息通道

在上一节的消息发布示例中，使用了 Source 组件默认提供的“output”通道。本节将不使用 Sink 组件默认提供的“input”通道，而是尝试通过自定义通道的方式来实现消息消费。在 Spring Cloud Stream 中，实现自定义通道的方法也非常简单，只需要定义一个新的接口，并在该接口中通过@Input 注解声明一个新的 Channel 即可。例如，可以定义一个新的 AccountChangeChannel 接口，然后通过@Input 注解就可以声明一个“inboundAccount Changes”通道，代码如下所示。注意，该通道的类型为 Spring Intergration 中用于消息消费的 SubscribableChannel。同时，整个 AccountChangeChannel 的代码风格与 8.2.1 节中的 Sink 接口完全一致。

```
import org.springframework.cloud.stream.annotation.Input;
import org.springframework.messaging.SubscribableChannel;

public interface AccountChangeChannel {

    @Input("inboundAccountChanges")
    SubscribableChannel accountChangeChannel();
}
```

3. 使用@StreamListener 消费消息

AccountChangeHandler 负责处理具体的消息消费逻辑，代码如下所示，这里有两个注解值得注意，一个是@EnableBinding 注解，一个是@StreamListener 注解。

```
@EnableBinding(AccountChangeChannel.class)
public class AccountChangeHandler {

    @Autowired
    private AccountRedisRepository accountRedisRepository;

private static final Logger logger =
        LoggerFactory.getLogger(AccountChangeHandler.class);

    @StreamListener("inboundAccountChanges")
    public void accountChangeSink(AccountChangeModel accountChange) {
```

```
            logger.debug("Received a message of type " +
                accountChange.getType());
            logger.debug("Received a {} event from the account service for
                account id {}", accountChange.getAction(),
                accountChange.getAccount().getId());

            if(accountChange.getAction().equals("ADD")) {
                accountRedisRepository
                .saveAccount(accountChange.getAccount());
            } else if(accountChange.getAction().equals("UPDATE")) {
                accountRedisRepository
                  .updateAccount(accountChange.getAccount());
            } else if(accountChange.getAction().equals("DELETE")) {
                accountRedisRepository
                  .deleteAccount(accountChange.getAccount().getId());
            } else {
                logger.error("Received an UNKNOWN event from the account
                  service of type {}", accountChange.getType());
            }
        }
    }
```

@EnableBinding 注解的作用是绑定一个或多个接口以实现消息的发送和接收。在上面的例子中，我们使用该注解绑定了自定义的 AccountChangeChannel。因为 AccountChange Channel 中通过@Input 注解提供了“inboundAccountChange”通道，所以这种用法实际上是和@EnableBinding(Source.class)以及@EnableBinding(Sink.class)是一致的。

@StreamListener 是新引入的注解，将该注解添加到某个方法上可以使之接收处理流中的事件。在上面的例子中，@StreamListener 注解添加在了 accountChangeSink()方法上并指向了“inboundAccountChange”通道，这意味着所有流经“inboundAccountChange”通道的消息都会由 accountChangeSink()方法进行处理。而在 accountChangeSink()方法中，调用 AccountRedisRepository 类完成各种缓存相关的处理。AccountRedisRepository 的实现代码参考如下，我们使用了 Spring Data 提供的 RedisTemplate 和 HashOperations 工具类来封装对 Redis 的数据操作。关于在 Spring Boot 应用程序中使用 Spring Data 的方法已在本书第 2 章中做了相应介绍。

```
@Repository
public class AccountRedisRepositoryImpl implements
    AccountRedisRepository {

    private static final String HASH_NAME ="account";

    private RedisTemplate<String, Account> redisTemplate;
```

仅供非商业用途或交流学习使用


```
    private HashOperations<String, Long, Account> hashOperations;

    @Autowired
    private AccountRedisRepositoryImpl(RedisTemplate<String, Account>
        redisTemplate) {
       this.redisTemplate = redisTemplate;
    }

    @PostConstruct
    private void init() {
        hashOperations = redisTemplate.opsForHash();
    }

    @Override
    public void addAccount(Account account) {
        hashOperations.put(HASH_NAME, account.getId(), account);
    }

    @Override
    public void updateAccount(Account account) {
        hashOperations.put(HASH_NAME, account.getId(), account);
    }

    @Override
    public void deleteAccount(Long accountId) {
        hashOperations.delete(HASH_NAME, accountId);
    }

    @Override
    public Account findAccountById(Long accountId) {
       return (Account) hashOperations.get(HASH_NAME, accountId);
    }
}
```

4. 使用消费者分组

就用户账户信息变更场景而言，可能存在这样的需求，即某些服务对 AccountChanged Event 事件的处理方式是放入缓存以便进行后续操作（如 Order 服务），而有些服务则可能会将该事件进一步转发到其他服务进行处理。这样对同一事件而言，不同的服务就会有不同的消费方式，即需要实现一对多的发布订阅模型。此外，服务多实例部署在现实的服务部署场景中非常常见，在集群环境下，我们希望服务的不同实例被放置在竞争的消费者关系中，同一服务只有一个实例能够处理给定消息。Spring Cloud Stream 提供的消费者分组可以很方便地实现这一需求。

仅供非商业用途或交流学习使用

消费者分组功能的结构示意图如图 8-17 所示。在图 8-17 中，AccountChangedEvent 事件会被同时发送给 OrderService 和 OtherService 集群，而在 OrderService 和 OtherService 集群内部，只有一个 OrderService 或 OtherService 实例能够消费该事件。

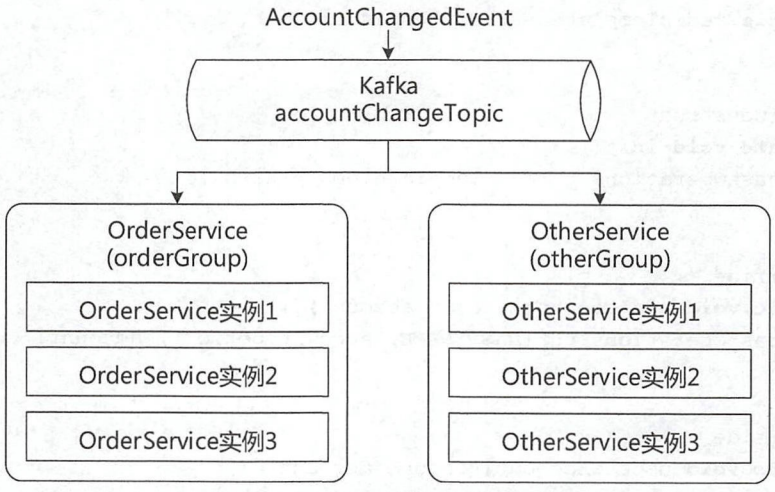


图 8-17　消费者分组结构示意图

要想实现如图 8-17 所示的消息消费效果，只需要在配置 Binder 时指定消费者分组信息即可，如下所示。

```
spring:
  cloud:
    stream:
      bindings:
        inboundAccountChanges:
          destination: accountChangeTopic
          content-type: application/json
          group: orderGroup
        kafka:
          binder:
            zkNodes: localhost
            brokers: localhost
```

以上配置信息中，“bindings”段的通道名称使用了自定义的“inboundAccountChanges”，并且在该配置项中设置了“group”为“orderGroup”。

仅供非商业用途或交流学习使用

8.5 本章小结

本章先从事件驱动架构与模型出发，给出了事件驱动架构的需求以及常见的基于消息传递机制实现事件驱动的基本思路。然后描述了事件驱动架构的基本模型以及与领域事件之间的关联。这部分内容是理解 Spring Cloud Stream 框架的基础。

与本书前面几章的行文思路不同，我们在引入 Spring Cloud Stream 时并不是直接介绍它的使用方法，而是从基本架构出发探讨该框架背后的实现原理。通过剖析 Spring Cloud Stream 背后的机制，可以发现它实际上是在企业集成服务框架 Spring Integration 以及 RabbitMQ、Kakfa 等多种主流消息中间件的基础上进行了抽象，并搭建的一个方便开发的整合层。

仅供非商业用途或交流学习使用

第 9 章

Spring Cloud Security 与服务安全

我们的系统可能时刻面临着各种潜在的网络攻击手段，应对安全攻击的基本手段可以分为两个不同的维度：一个是网络架构，通常表现为使用内外网分离、防火墙、网闸等手段或设备；另一个则是关注于应用程序本身，即在软件实现上保证其安全性。本章主要关注的是应用程序这一维度。

常见的应用程序级别的安全控制手段包括信息加密、认证、授权和安全性协议等。信息加密的技术主要包括单向散列加密、对称加密和非对称加密 3 种，单向散列（Hash）加密能够将不同长度的明文转换成固定长度的密文，且转化过程单向不可逆；对称加密中加密和解密使用同一秘钥，而非对称加密中加密和解密分别使用公钥和私钥机制。认证（Authentication）和授权（Authorization）的含义比较容易混淆，认证的目的在于回答“你是谁”这一问题，常见的有摘要认证和签名认证；而授权的目的是在“你是谁”之后进一步明确“你能做什么”，通用的授权模型是对资源、权限、角色和用户的组合。在协议层面也有一些成熟方案，如 HTTPS（HTTP over Secure Socket Layer）协议能将整个通信过程进行加密。

微服务架构作为面向互联网的分布式服务架构，同样需要考虑各种应用级别的安全性实现手段。但在微服务环境下，安全性的实现也会面临特定场景下的挑战。因为在微服务架构中，各服务部署在分布式环境中的多套容器之内。各服务的接口不是存在于本地，而是通过 HTTP 进行远程接入。这里所说的挑战主要包括以下两个方面：

- 如何验证用户并在不同微服务之间完成认证身份信息的传递
- 如何让各个微服务完成对用户的授权

本章内容将围绕这两个问题具体展开，引出微服务架构中常用的 OAuth 协议以及实现框架 Spring Cloud Security，并给出相应的案例实现。

9.1 服务访问安全性与 OAuth 协议

微服务架构的安全性本质上指的就是服务访问的安全性。本节将介绍服务访问安全性设计的常用策略，并引出基于 Token（令牌）的认证方案。同时，业界也有成熟的协议可直接用于微服务之间的安全访问，这种协议就是 OAuth 协议，本节会对 OAuth 协议的特点及实现方案

仅供非商业用途或交流学习使用

进行详细展开。

9.1.1 微服务架构中的安全性设计

在微服务架构下，我们设想一下服务访问过程中需要考虑的安全性问题。首当其冲的是要考虑外部应用的接入，同时也要面临用户到服务的鉴权、服务到服务的鉴权等多种鉴权场景。在介绍具体方案和工具之前，我们需要设想一下完成这些场景的大致思路。

1. 微服务安全策略

基于以往在分布式环境下处理 Session 的场景，分布式 Session 是一种可以类比的方案。分布式 Session 方案的原理是，把关于用户认证的信息存储在共享存储中，且通常将用户会话作为 Key 来实现简单分布式哈希映射。当用户访问微服务时，用户数据可以从共享存储中获取。在某些场景下，这种方案很不错，业界也有很多成熟的工具可用于实现对分布式 Session 的管理。

基于分布式 Session 解决方案存在的主要问题是服务器端保存了用户认证的状态信息。笔者在《微服务设计原理与架构》中提到服务无状态是实施微服务架构的一项最佳实践。为了避免将状态信息维护在服务器端，我们可以切换思路，将状态信息维护在客户端。在这种思路下产生的解决方案就是客户端 Token 机制。基于 Token 认证的好处就在于服务端无状态，Token 机制在服务端不需要存储 Session 信息，因为 Token 自身包含了用户的所有相关信息。同时，客户端 Token 机制也具备很好的扩展性。

本书第 6 章介绍了 API 网关，实际上网关的一大作用就是进行安全控制，所以我们可以把客户端 Token 与 API 网关结合起来。这种方案意味着所有请求都通过网关，从而有效地隐藏了微服务。基于 Token 的实现方案在服务访问场景下应用非常广泛，在介绍具体的实现方法之前，我们先对这一方案做进一步介绍。

2. 基于 Token 的认证方案

随着 RESTful API 和微服务的兴起，基于 Token 的认证现在已经越来越普遍。Token 和 Session ID 不同，并非只是一个 Key。Token 一般会包含用户的身份信息，通过验证 Token 就可以完成身份校验。图 9-1 展示了基于 Token 的基本工作机制。

图 9-1 所示的基于 Token 的验证流程如下所示。

① 用户使用包含用户名和密码的身份凭证从客户端发起资源请求。

② 后端接受请求，通过授权中心生成有效 Token 字符串，并返回给客户端。

③ 客户端获得 Token 后，再次发出资源请求。

④ 后端接受带 Token 的请求，通过授权中心获取相关资源并返回给客户端。

仅供非商业用途或交流学习使用

基于 Token 的认证只是一种方案而不是具体的技术，业界关于如何实现该方案也形成了一些特定的工具和协议，最具代表性的就是 OAuth 协议。

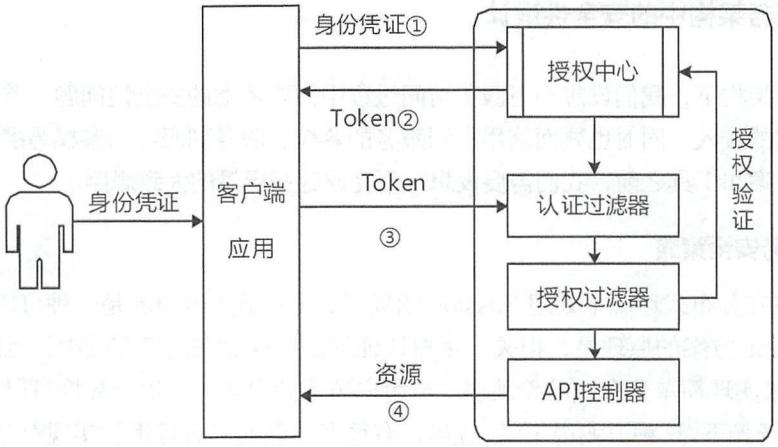


图 9-1　Token 机制基本示意图

9.1.2　OAuth 协议

OAuth（Open Authorization）协议是面向 SOA 和 ESB 等分布式场景下开放和消费第三方接口，该协议解决的是授权问题而不是认证问题。

1. OAuth 协议应用场景

为了理解 OAuth 协议的适用场景，下面来举一个例子。在移动医疗场景下，我们要开发一个中医预约推荐平台，该平台可以根据用户存储在健客网（www.jianke.com，一家在线药品电商网站）中的药品购买记录自动推荐相关的中医专家。用户为了使用该服务，必须让中医预约推荐平台读取自己在健客网上的药品购买记录。

在这个场景中，难点在于只有得到用户的授权，健客网才会同意中医预约推荐平台读取用户的药品购买记录。那么，中医推荐平台如何获得用户的授权呢？一种方法是用户将自己的健客网用户名和密码告诉中医推荐平台，后者就可以通过用户名和密码登录到健客网并读取用户的购药记录，但这样做存在几个严重的缺点。

中医推荐平台为了后续的服务，会保存用户在健客网上的密码，这样很不安全；中医推荐平台拥有了获取用户存储在健客网上所有资料的权力，用户没法限制中医推荐平台获得授权的范围和有效期；用户只有修改密码，才能收回赋予中医推荐平台的权力，但是这样做会使得其他所有获得用户授权的类似中医推荐平台的第三方应用程序全部失效；只要有一个第三方应用程序被破解，就会导致用户密码泄漏，以及所有被密码保护的数据产生泄漏。OAuth 协议的诞

仅供非商业用途或交流学习使用

生就是为了解决以上问题。

2. OAuth 协议流程

在详细讲解 OAuth 2.0 协议之前，需要说明几个专用名词。

- Third-party Application

第三方应用程序，也可以称为客户端，即上文例子中的中医推荐平台。

- HTTP Service

HTTP 服务提供商，即上文例子中的健客网。

- Resource Owner

资源所有者，即上文例子中的用户。

- User Agent

用户代理，很多场合指的是浏览器。

- Authorization Server

认证服务器，即服务提供商专门用来处理认证的服务器。

- Resource Server

资源服务器，即服务提供商存放用户生成资源的服务器。它与认证服务器可以是同一台服务器，也可以是不同的服务器。

OAuth 协议的作用就是让客户端安全可控地获取用户的授权，并与服务提供商进行互动。OAuth 协议在客户端与服务提供商之间设置了一个授权层（Authorization Layer）。客户端不能直接登录服务提供商系统，而是只能登录授权层，客户端登录授权层所用的 Token 与用户密码不同，以此将用户与客户端区分开来。用户可以在登录的时候，指定授权层 Token 的权限范围和有效期。客户端登录授权层以后，服务提供商根据 Token 的权限范围和有效期向客户端开放用户存储的资源。OAuth 2.0 的基本运行流程如图 9-2 所示，该图来自 OAuth 协议的规范文档 RFC 6749。

图 9-2 展示了 OAuth 协议的主要步骤。

① 用户打开客户端以后，客户端要求用户给予授权。

② 用户同意给予客户端授权。

③ 客户端使用上一步获得的授权，向认证服务器申请 Token。

④ 认证服务器对客户端进行认证，确认无误后同意发放 Token。

⑤ 客户端使用 Token，向资源服务器申请获取资源。

⑥ 资源服务器确认 Token 无误后，同意向客户端开放资源。

仅供非商业用途或交流学习使用

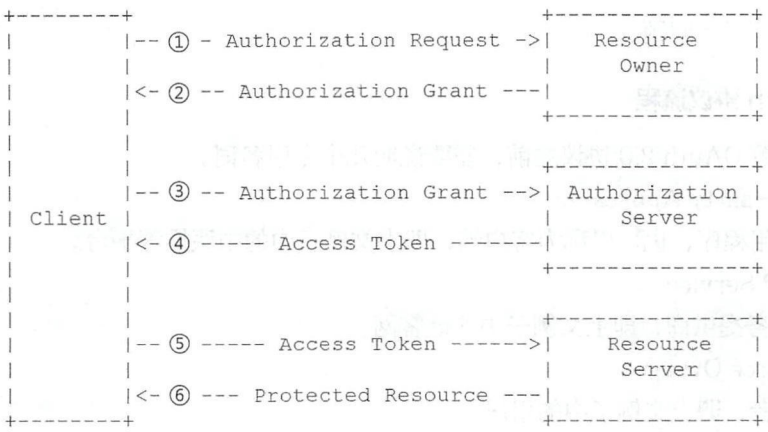


图 9-2　OAuth 协议时序图

把上述流程应用到微服务架构的场景中，显然作为服务提供者，其所充当的角色就是资源服务器，而服务调用者就是客户端。所以各个服务本身既可以是客户端，又可以作为资源服务器，或者两者兼之。当客户端拿到 Token 之后，该 Token 就能在各个服务之间进行传递。对图 9-2 进行简化并重新应用到服务访问的场景，可以得到如图 9-3 所示的 OAuth 协议运行效果图。

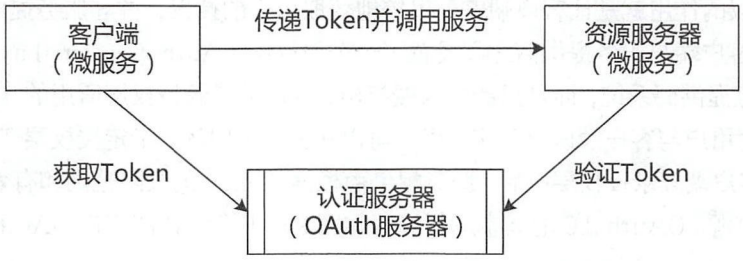


图 9-3　OAuth 在服务访问场景中的应用

在图 9-3 中，可以看到对于那些需要确保访问得到合理控制的服务而言，我们需要为其构建资源服务器，资源服务器管理着能够访问该服务的应用和用户信息；对于需要访问那些受资源服务器保护的微服务而言，必须要进行认证并获取 Token；而位于资源服务器和客户端之间的就是认证服务器，该认证服务器同时需要完成对用户进行认证以及对客户端提交的 Token 进行验证。

3. OAuth 授权模式

在整个 OAuth 协议中，有一步是非常关键的，即用户如何才能获取客户端授权。有了这个授权以后，客户端就可以获取 Token，进而凭 Token 获取资源。OAuth 2.0 定义了 4 种授权方

仅供非商业用途或交流学习使用

式，即授权码模式（Authorization Code）、简化模式（Implicit）、密码模式（Resource Owner Password Credentials）和客户端模式（Client Credentials）。本书无意对这4种授权方式一一详细展开，读者如果想要了解这些授权模式，可以参考相关资料。下面将简单介绍授权码模式和密码模式，在后续的服务构建环节，密码模式将是我们用来展示工具和代码的主要授权模式。

授权码模式功能最完整，流程也最严密，该模式的特点是能通过客户端系统的后台服务器，与服务提供商的认证服务器进行交互。图9-4展示了授权码模式的典型时序图。

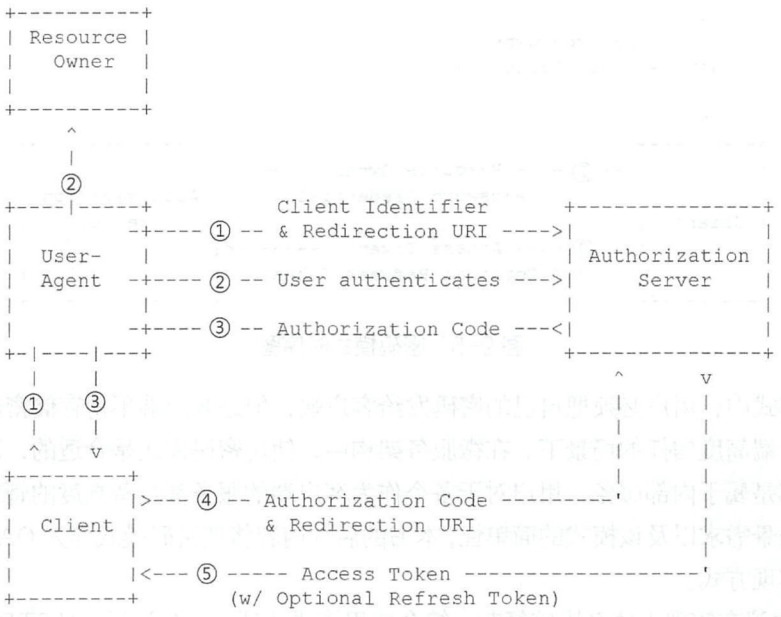


图9-4 授权码模式时序图

图9-4中的几个步骤说明如下。

① 用户访问客户端，后者将前者导向认证服务器。

② 用户选择是否给予客户端授权。

③ 假设用户给予授权，认证服务器将用户导向客户端事先指定的重定向URI（Redirection URI），同时附上一个授权码。

④ 客户端收到授权码，附上早先的重定向URI，向认证服务器申请Token。这一步是在客户端系统的后台服务器上完成的，对用户不可见。

⑤ 认证服务器核对授权码和重定向URI，确认无误后，向客户端发送Access Token（访问令牌）和Refresh Token（更新令牌）。

相较授权码模式，密码模式则比较简单，也更加容易理解。密码模式的工作流程如图9-5所示，几个主要步骤如下所示。

① 用户向客户端提供用户名和密码。

② 客户端将用户名和密码发给认证服务器，向后者请求 Token。

③ 认证服务器确认无误后，向客户端提供 Access Token。

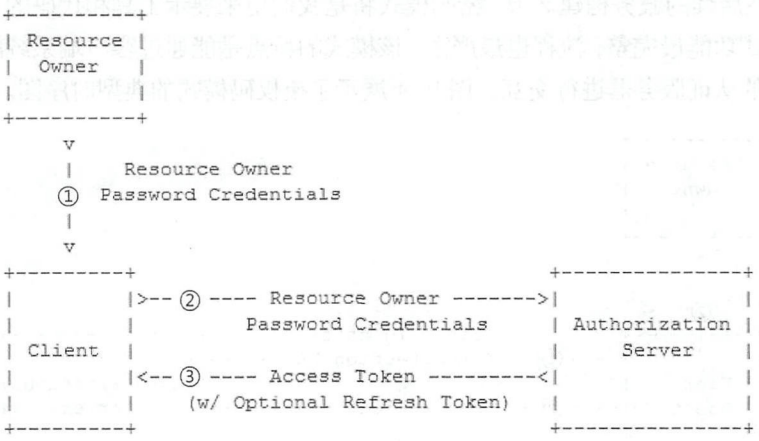


图 9-5 密码模式时序图

在密码模式中，用户必须把自己的密码发给客户端，但是客户端不得存储密码，这通常用在用户对客户端高度信任的场景下。在微服务架构中，使用密码模式是合适的，因为所有的微服务原则上都是属于内部服务，用户对于各个作为客户端的服务都具备高度的管理能力。正因为满足这一场景需求以及该模式的简单性，本书的后续内容将把密码模式作为 OAuth 协议授权模式的默认实现方式。

OAuth 协议在实现上体系比较复杂，综合应用摘要认证、签名认证、HTTPS 等安全性手段，需要提供 Token 生成和校验、分布式 Session 和公私钥管理等功能，同时需要开发者入驻并进行权限粒度控制。一般我们应该避免自己去实现如此复杂的协议，而是倾向于借助特定工具以避免重复造轮子。下面将引入 Spring Cloud 家族中的另一个重要成员 Spring Cloud Security 来完成 OAuth 服务认证流程的实现。

9.2 构建 OAuth 认证服务器

在图 9-3 中，可以看到位于资源服务器和客户端之间的就是认证服务器，该认证服务器同时需要完成对用户进行认证以及对客户端提交的 Token 进行验证。当用户每次访问受保护的服务之前都需要与认证服务器进行交互，以确认该用户是否拥有合法的 Token 以及该 Token 是否已经过期。本节的重点就是讨论如何构建认证服务器，从而为下一节中的服务访问控制提供基础。

9.2.1 引入 Spring Cloud Security

在微服务架构的实现过程中，认证服务器和注册中心服务器、配置服务器一样也表现为一个微服务，因此构建认证服务器的方法也是创建一个 Spring Boot 应用程序，我们需要引入合适的 maven 依赖以及提供一个 Bootstrap 类作为访问的入口。

1. Spring Cloud Security 与 Maven 依赖

在第 8 章的基础上，我们将在整个代码工程中创建一个新的 maven 工程并取名为 authentication-server。为了构建认证服务器，还需要引入 OAuth 相关的 jar 包依赖，如下所示。其中，第一个依赖 spring-cloud-security 属于 Spring Cloud 家族中的一员，而 spring-security-oauth2 则是来自 Spring Security 中的 OAuth2 库。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-security</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

作为 Spring Cloud 中的一员，Spring Cloud Security 是对微服务架构中所面临的安全性问题进行抽象并实现的工具。Spring Cloud Security 具备众多特点，包括基于 OAuth2 和 OpenID 协议的可配置的 SSO 登录机制；基于 Token 保障资源访问安全；引入 UAA(User Account and Authentication，用户账户和认证)服务。其中，UAA 是一个 Web 服务，主要功能是封装 OAuth2 协议，用于管理用户账户、OAuth2 客户端以及用户用于鉴权的 Token。

当用户访问客户端应用时，UAA 会生成并发放 Token 给目标客户端。结合 9.1.2 节中介绍的 OAuth 协议，UAA 在以下几个方面对 OAuth 协议做了很好的封装。

- 认证对象

在认证对象上，UAA 支持用户、客户端及目标资源服务器。

- 认证类型

UAA 中的认证类型主要有授权码模式、密码模式及客户端模式。

- 认证范围

在认证范围上，UUA 是通过一个命名的参数附加到 Token 上的方式进行实现。

2. 使用@EnableAuthorizationServer 注解

现在 maven 依赖已经添加完毕，下一步就是构建 Bootstrap 类。以下代码位于 authentication-server 工程的 AuthenticationApplication 类中，展示了其作为 Bootstrap 类的代码结构。

```
@SpringBootApplication
@RestController
@EnableEurekaClient
@EnableAuthorizationServer
public class AuthenticationApplication {

    public static void main(String[] args) {
        SpringApplication.run(AuthenticationApplication.class, args);
    }
}
```

以上代码中值得注意的是一个新的注解，即@EnableAuthorizationServer 注解。@EnableAuthorizationServer 注解的作用在于为微服务运行环境提供一个基于 OAuth 协议的认证服务。

启动 AuthenticationApplication 类时，会发现在控制台日志中有如下信息，代表着一系列以/oauth 为根地址的 RESTful 风格的端点已经启动并处于监听状态。

```
s.o.p.e.FrameworkEndpointHandlerMapping : Mapped "{[/oauth/authorize]}"…
s.o.p.e.FrameworkEndpointHandlerMapping : Mapped "{[/oauth/authorize],
methods=[POST],params=[user_oauth_approval]}" …
s.o.p.e.FrameworkEndpointHandlerMapping : Mapped "{[/oauth/token],methods=
[POST]}" …
s.o.p.e.FrameworkEndpointHandlerMapping : Mapped "{[/oauth/token],methods=
[GET]}" …
s.o.p.e.FrameworkEndpointHandlerMapping : Mapped "{[/oauth/check_token]}" …
s.o.p.e.FrameworkEndpointHandlerMapping : Mapped "{[/oauth/confirm_access]}"
…
s.o.p.e.FrameworkEndpointHandlerMapping : Mapped "{[/oauth/error]}" …
```

9.2.2 初始化用户与客户端

上一小节通过引入 Spring Cloud Security 构建了认证服务器，认证服务器是一种集中式系统，管理着所有与认证相关的客户端和用户信息。因此，我们需要在认证服务器中对这些基础信息进行初始化。为了实现这一目标，当使用@EnableAuthorizationServer 注解来嵌入 OAuth

授权服务机制时，可以通过使用框架提供的各种配置类来配置这个授权服务。

1. 初始化用户

初始化用户操作所依赖的配置类是 WebSecurityConfigurer 类，Spring Security 同样提供了 org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurer-Adapter 类来简化该配置类的使用方式，我们可以继承 WebSecurityConfigurer Adapter 类并且覆写其中的 configure()方法来完成配置工作。

关于 WebSecurityConfigurer 配置类，我们首先需要明确配置的内容。实际上，初始化用户所使用的用户信息非常简单，只需要指定用户名(Username)、密码(Password)和角色(Role)这三项数据即可。在 WebSecurityConfigurer 类中，使用 AuthenticationManagerBuilder 类来创建一个 AuthenticationManager 能够轻松实现基于内存、LADP 和 JDBC 的验证。本例使用 AuthenticationManagerBuilder 类中的 inMemoryAuthentication()方法来实现基于内存的用户信息认证。示例代码如下所示。

```
@Override
protected void configure(AuthenticationManagerBuilder builder) throws
    Exception {

    builder.inMemoryAuthentication()
           .withUser("tianyalan_user").password("password1")
           .roles("USER")
           .and()
           .withUser("tianyalan_admin").password("password2")
           .roles("USER", "ADMIN");
}
```

从上面的代码中，可以看到系统中存在“tianyalan_user”和“tianyalan_admin”这两个用户，其密码分别是“password1”和“password2”，在角色上也分别代表着普通用户 USER 及管理员 ADMIN。关于用户信息的初始化，我们也创建了一个 WebSecurityConfigurer 类来继承 WebSecurityConfigurerAdapter 类并覆写相应的 configure()方法，其完整代码如下所示。

```
@Configuration
public class WebSecurityConfigurer extends WebSecurityConfigurerAdapter {

    @Override
    @Bean
    public AuthenticationManager authenticationManagerBean() throws
       Exception {

       return super.authenticationManagerBean();
    }
```

```
    @Override
    @Bean
    public UserDetailsService userDetailsServiceBean() throws
       Exception {

       return super.userDetailsServiceBean();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder builder)
        throws Exception {

        builder.inMemoryAuthentication()
               .withUser("tianyalan_user").password("password1")
               .roles("USER")
               .and()
               .withUser("tianyalan_admin").password("password2")
               .roles("USER", "ADMIN");
    }
}
```

在上述代码中，可以看到有两个 authenticationManagerBean()和 userDetailsServiceBean()方法分别返回了父类的默认实现，这里返回的 UserDetailsService 和 AuthenticationManager 在初始化客户端时会再次用到。

2. 初始化客户端

初始化客户端时，首先使用到的配置类是 ClientDetailsServiceConfigurer，该配置类用来配置客户端详情服务 ClientDetailsService，涉及安全认证相关的客户端信息都将在该类中得到初始化。ClientDetailsService 中有如下几个重要的属性。

- clientId

必填属性，用于标识客户的 Id。

- secret

用于设置客户端安全码。

- scope

用于设置客户端的访问范围，默认为空。如果为空，客户端就拥有全部的访问范围。常见的设置方式包括 webclient 和 mobileclient，分别代表 Web 端和移动端。

- authorizedGrantTypes

客户端可以使用的授权模式有 5 种：authorization_code，代表授权码模式；implicit，代表隐式授权模式；password，代表密码模式；client_credentials，代表客户端凭据模式；

refresh_token，通过刷新操作获取以上授权模式下所产生的新的令牌。

Spring Security 提供了 org.springframework.security.oauth2.config.annotation.web.configuration. AuthorizationServerConfigurerAdapter 类来简化配置类的使用方式，通过继承AuthorizationServerConfigurerAdapter并覆写其中的configure()方法来进行配置即可使配置类生效。使用 AuthorizationServerConfigurerAdapter 进行客户端信息配置的基本代码结构如下。

```
@Configuration
public class AuthorizationServerConfigurer extends
    AuthorizationServerConfigurerAdapter {

    @Autowired
    private UserDetailsService userDetailsService;

    @Override
    PUBLIC Void configure(ClientDetailsServiceConfigurer clients)
        throws Exception {

    }
}
```

初始化客户端信息有两种方式，一种是直接把客户端信息写在代码中以实现内存级别的存储，另一种是通过 JDBC 在数据库中存储详情信息。为了简便起见，本书使用.inMemory()方法实现内存级别的存储方式以初始化客户端信息，示例代码如下。可以看到我们分别设置了客户Id为“tianyalan”、客户端安全码为“secret”以及三种授权模式，最后把客户端的访问范围设为同时支持 Web 端和移动端。

```
clients.inMemory()
       .withClient("tianyalan")
       .secret("secret")
       .authorizedGrantTypes("refresh_token", "password",
            "client_credentials")
       .scopes("webclient", "mobileclient");
```

授权是使用 AuthorizationEndpoint 这个端点来进行控制的，要想对该端点的行为进行配置，可以使用 AuthorizationServerEndpointsConfigurer 配置类。和 ClientDetails Service Configurer 配置类一样，我们也通过继承 AuthorizationServerConfigurerAdapter 并且覆写其中的 configure()方法来进行配置。AuthorizationServerEndpointsConfigurer 配置类中可以设置的属性有以下几项。

- userDetailsService

如果设置了这个属性，那就代表着使用一个自定义的 UserDetailsService 接口实现去替

仅供非商业用途或交流学习使用

换全局的实现。一旦该属性被设置，那么“refresh_token”授权模式的流程中就会包含一个检查，用来确保这个账号是否仍然有效。

- authenticationManager

认证管理器，当选择了密码授权模式时，需要设置这个属性以注入一个 AuthenticationManager 对象。

- authorizationCodeServices

用于设置授权码模式。

- implicitGrantService

用于设置隐式授权模式。

上文中我们设置的授权模式是密码授权模式，所以需要在代码中指定 userDetailsService 和 authenticationManager 以配置 AuthorizationServerEndpointsConfigurer，示例代码如下。

```
@Autowired
private AuthenticationManager authenticationManager;

@Autowired
private UserDetailsService userDetailsService;

@Override
public void configure(AuthorizationServerEndpointsConfigurer endpoints)
    throws Exception {

    endpoints.authenticationManager(authenticationManager)
            .userDetailsService(userDetailsService);
}
```

回想我们在初始化用户中生成的 UserDetailsService 和 AuthenticationManager，不难理解这两个类被用来注入 AuthorizationServerEndpointsConfigurer 中去配置相应的端点信息。

至此，客户端初始化工作全部完成，下面将创建一个 AuthorizationServerConfigurer 类来继承 AuthorizationServerConfigurerAdapter 类并覆写相应的 configure()方法，完整的 AuthorizationServerConfigurer 类代码如下所示。

```
@Configuration
public class AuthorizationServerConfigurer extends
AuthorizationServerConfigurerAdapter {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private UserDetailsService userDetailsService;
```

仅供非商业用途或交流学习使用

```
    @Override
    public void configure(AuthorizationServerEndpointsConfigurer
        endpoints) throws Exception {

        endpoints.authenticationManager(authenticationManager)
                .userDetailsService(userDetailsService);
    }

    @Override
    public void configure(ClientDetailsServiceConfigurer clients)
        throws Exception {

        clients.inMemory()
                .withClient("tianyalan")
                .secret("secret")
                .authorizedGrantTypes("refresh_token", "password",
                    "client_credentials")
                .scopes("webclient", "mobileclient");
    }
}
```

9.2.3 生成 Token

9.2.1 节中我们引入了 Spring Cloud Security 并构建了 AuthenticationApplication 类。9.2.2 节中，我们也通过初始化用户和客户端信息为授权的 AuthorizationServerEndpoints Configurer 类提供了相应的配置，接下来就可以利用这些配置信息来实现对用户的认证。本节将通过 Postman 来模拟服务的请求并获取 Token 和用户信息，这是实现服务访问控制的前提条件。

1. 获取 Token

当认证服务器启动完毕，就可以使用 http://localhost:8080/oauth/token 端点来获取 Token。在使用该端点时，需要提供两部分数据，即在上一节中所配置的客户端信息和用户信息。

在 Postman 中，客户端信息设置方式如图 9-6 所示。在“Authorization”请求头中指定认证类型为“Basic Auth”，然后设置客户端名称和客户端安全码分别为“tianyalan”和“secret”。

接下来指定针对授权模式的专用配置信息。

- grant_type

该属性代表 OAuth 的授权模式，本书将使用密码模式来演示认证过程，所以该属性值为

仅供非商业用途或交流学习使用

“password”。

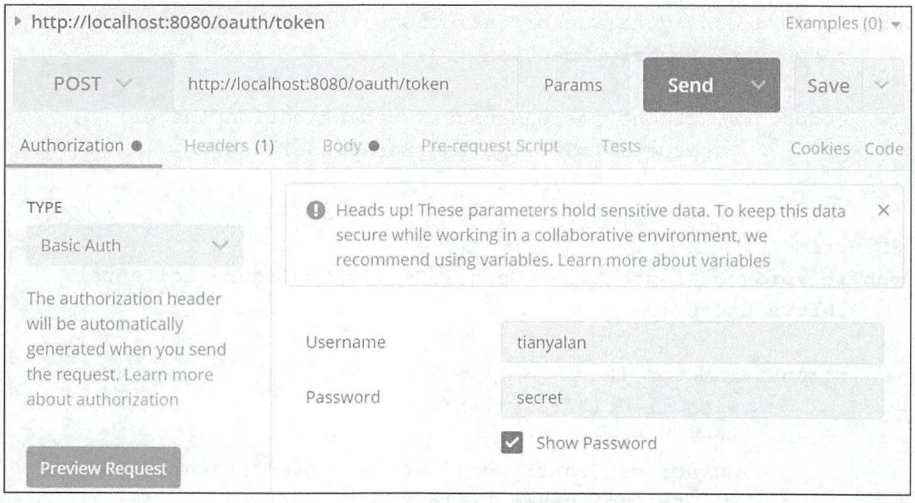


图 9-6　设置客户端信息示意图

- scope

用于指定客户端的访问范围。在初始化客户端时 scope 属性被设置为“webclient”和“mobileclient”，所以只能在这两个属性值中选择一个，这里使用“webclient”。

- username 和 password

用于设置用户名和密码以识别用户身份。我们先使用“tianyalan_user”这个用户进行认证，请记住该用户的角色是“USER”，而不是“ADMIN”。

在使用 Postman 提交 POST 请求时，以上 4 个参数将作为表单（Form）的一部分发送到服务器，用户信息设置效果如图 9-7 所示。

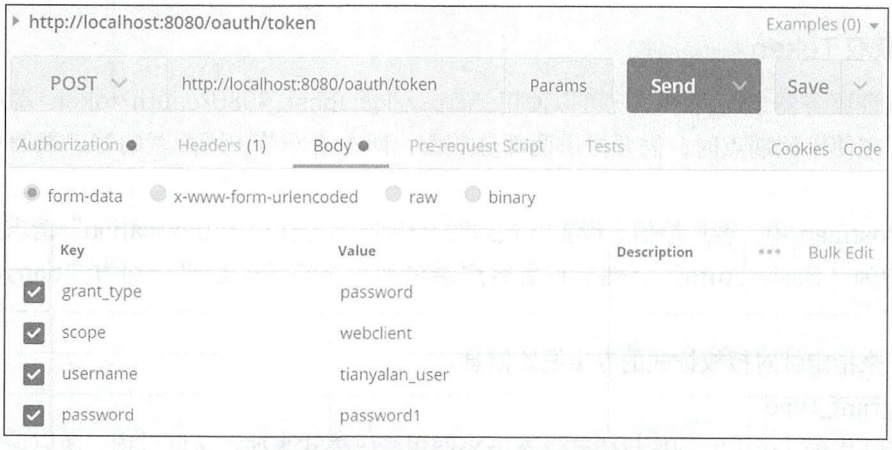


图 9-7　设置用户信息示意图

仅供非商业用途或交流学习使用

图 9-8 展示了 http://localhost:8080/oauth/token 端点的返回结果，除了作为请求参数的 scope，我们可以看到在消息体中包含了以下信息。

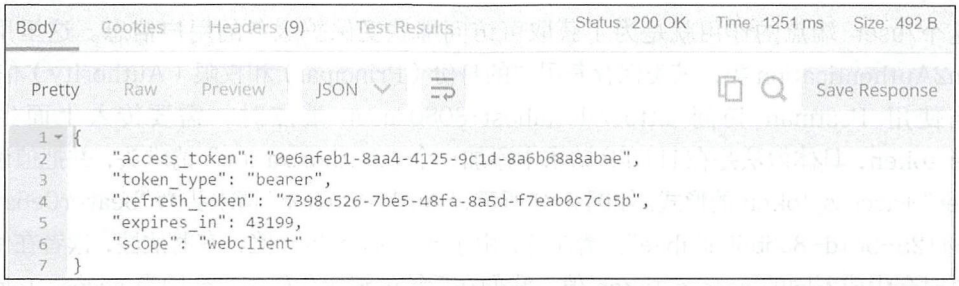


图 9-8　返回结果示意图

- access_token

access_token 就是 OAuth token，当访问每个受保护的服务时，用户都需要携带该 Token 以便进行验证。为了简便起见，如果不加特殊说明，下文中提到的 Token 指的就是这个 access_token。

- token_type

在 OAuth 协议中存在多种 Token 类型可供选择，包括 bearer 类型、mac 类型等，这里返回的是最常见的一种类型，即 bearer 类型。

- refresh_token

当 access_token 过期之后，refresh_token 用于重新下发一个新的 access_token。refresh_token 会自动传回认证服务器，我们一般不需要关注。

- expires_in

expires_in 属性用于指定 access_token 的有效时间，当超过这个有效时间时，access_token 将会自动失效。

至此，我们已经成功获取了可用于访问各个服务的 Token 信息，下一节将具体演示如何使用该 Token。在这之前，我们也可以通过 Token 获取当前的认证用户信息，这在有些场景下非常有用。

2. 获取用户信息

为了获取用户信息，可在 AuthenticationApplication 类中添加一个新的端点，代码如下。

```
@RequestMapping(value = { "/user" }, produces = "application/json")
public Map<String, Object> user(OAuth2Authentication user) {
    Map<String, Object> userInfo = new HashMap<>();
    userInfo.put("user", user.getUserAuthentication().getPrincipal());
    userInfo.put("authorities", AuthorityUtils.authorityListToSet(
```

仅供非商业用途或交流学习使用

```
            user.getUserAuthentication().getAuthorities()));
        return userInfo;
    }
```

这个/user 端点的作用就是为了获取可访问那些受保护服务的用户信息。这里用到了OAuth2Authentication 类，该类保存着用户的身份（Principal）和权限（Authority）信息。

当使用 Postman 访问 http://localhost:8080/user 端点时，需要传入上面生成的access_token，具体做法是在 HTTP 请求中添加一个“Authorization”请求头，并把值设置成“bearer”+access_token 的形式。如图 9-9 所示，“Authorization”请求头为“bearer0e6afeb1-8aa4-4125-9c1d-8a6b68a8abae”。请注意，由于 access_token 值具有时效性，读者在使用示例代码时会生成不同的 access_token 值，我们在后续的演示代码中也会因为 access_token 的过期而重新生成新的 access_token。

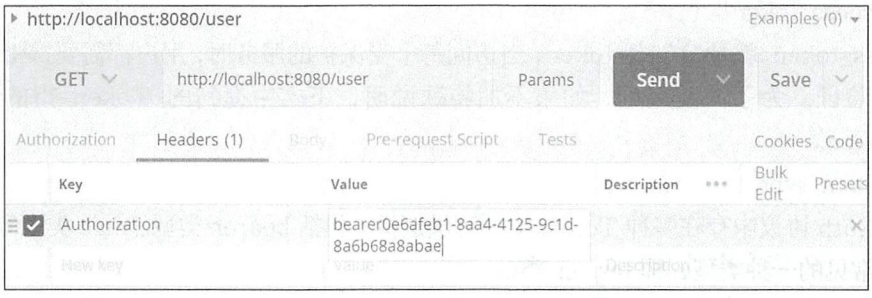


图 9-9　获取用户信息请求示意图

图 9-10 展示了该请求的结果，可以看到该结果中包含了用户的用户名（tianyalan_user）、密码（考虑到安全性不显示）以及角色（ROLE_USER），这些信息与我们在 9.2.2 节中所初始化的用户信息保持一致。

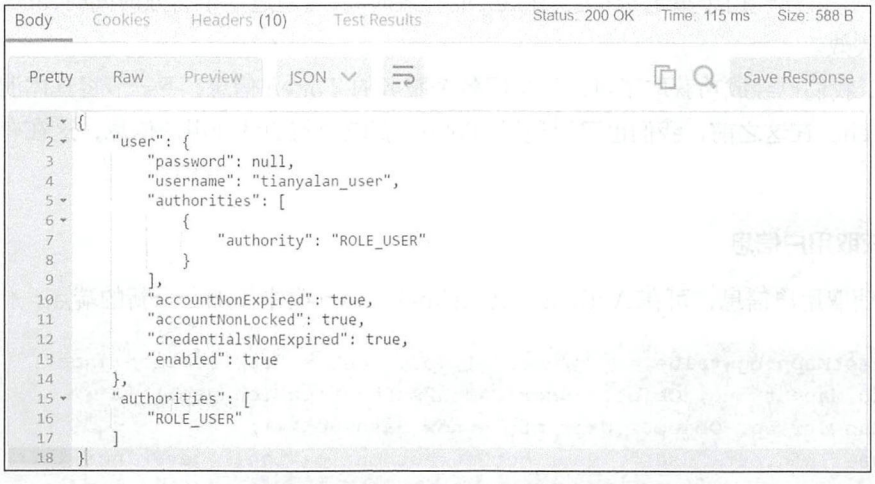


图 9-10　获取用户信息结果示意图

仅供非商业用途或交流学习使用

至此，已成功构建了 OAuth 认证服务器，下一节将从各个微服务角度出发，讨论如何利用 OAuth 认证服务器保护服务之间的相互访问。

9.3 使用 OAuth 保护服务访问

从服务访问角度讲，使用 OAuth 保护服务访问需要解决以下两个问题。

- 单个服务的访问策略控制

对于单个微服务而言，需要根据服务的重要性设置不同的访问策略以实现灵活的权限控制。

- 多个服务链路上的服务访问传播

当使用 Zuul 构建服务访问链路时，需要确保 Token 在整个访问链路上的有效传播。

围绕这两个问题，本节首先介绍如何在单个微服务中集成 OAuth 认证服务，然后讨论在单个服务中创建服务访问的 3 种不同策略，最后使用 OAuth2RestTemplate 这一特定工具类完成 Token 在服务链路中的传播。

9.3.1 集成 OAuth 认证服务

现在我们切换角度，回到单个微服务。请注意，在 OAuth 协议中，单个微服务的角色是受保护的资源。在服务调用时，所有发往单个微服务的请求都应该包含 OAuth Token 信息，然后单个微服务再根据这个 Token 信息请求认证服务器以验证该 Token 是否有效。

1. 配置认证服务器

为了实现单个微服务与认证服务器之间的交互，我们需要在单个微服务中配置认证服务器信息。以 Product 服务为例，与 authentication-server 一样，我们首先需要确保项目工程中添加了 spring-cloud-security 和 spring-security-oauth2 这两项 maven 依赖。然后需要在 application.yml 配置文件中添加对认证服务器中的用户信息端点 http://localhost:8080/user 的引用，具体涉及的配置项为 security.oauth2.resource.userInfoUri，示例如下。

```
security:
  oauth2:
    resource:
      userInfoUri: http://localhost:8080/user
```

通过 security.oauth2.resource.userInfoUri 配置项，就可以与认证服务器进行交互，从而在单个微服务和认证服务器之间建立一种调用关系。

仅供非商业用途或交流学习使用

2. 使用@EnableResourceServer 注解

在 OAuth 协议中，单个微服务的定位就是资源服务器（Resource Server），保存着需要限制访问的资源。Spring Security 框架为此提供了专门的@EnableResourceServer 注解。通过在 Bootstrap 类中添加@EnableResourceServer 注解，相当于声明了该服务中的所有内容都是受保护的资源。以 ProductApplication 类为例，示例代码如下。

```
@SpringBootApplication
@EnableEurekaClient
@EnableCircuitBreaker
@EnableResourceServer
public class ProductApplication {

    public static void main(String[] args) {
        SpringApplication.run(ProductApplication.class, args);
    }
}
```

在添加了@EnableResourceServer 注解之后，单个微服务会对所有的 HTTP 请求进行验证以确定消息头中是否包含 Token 信息，如果没有 Token 信息，则会直接限制访问；如果有 Token 信息，就会通过 security.oauth2.resource.userInfoUri 配置项访问认证服务器并进行 Token 的验证。

9.3.2 创建服务访问策略

作为资源服务器，每个微服务对于自身资源的保护粒度因服务而异。这里的资源在 Spring Cloud 中实际上指的就是 RESTful 风格下的 URL 和相应的 HTTP 操作。对于一般的资源，可能只需要用户通过认证就都可以访问；而对于一些重要资源，则在用户已认证的基础上还会有附加的要求。本节讨论的就是对资源进行保护的级别，具体包括以下 3 种。

- 认证用户级别

该级别是最基本的资源保护级别，只要是认证用户就可能访问服务内的各种资源。

- 认证用户+角色级别

该级别在认证用户级别的基础上，还要求用户属于某一个或多个特定角色。

- 认证用户+角色+操作级别

该级别在认证用户+角色级别的基础上，还对某些 HTTP 操作做了访问限制。

1. 使用认证用户级别保护服务访问

在 9.2.2 节中，我们已经熟悉了通过扩展各种 ConfigurerAdapter 类来实现自定义配置信

仅供非商业用途或交流学习使用

息的方法。对于资源服务器而言，也存在一个 ResourceServerConfigurerAdapter 类，通过继承该类并覆写其 configure()方法，就可以对资源的访问规则做相应的配置。为此，我们创建了一个 ResourceServerConfiguration 类，示例代码如下。

```
@Configuration
public class ResourceServerConfiguration extends
    ResourceServerConfigurerAdapter {

    @Override
    public void configure(HttpSecurity http) throws Exception{
        http.authorizeRequests()
            .anyRequest()
            .authenticated();
    }
}
```

位于 configure()方法中的 http.authorizeRequests().anyRequest().authenticated();语句指定了访问该服务的任何请求都需要进行验证。因此，当我们使用普通的 HTTP 请求来访问 Product 服务中的任何 URL（如 http://localhost:8081/v1/products/001）时，将会得到如图 9-11 所示的错误信息，该错误信息明确指出资源的访问需要进行认证。

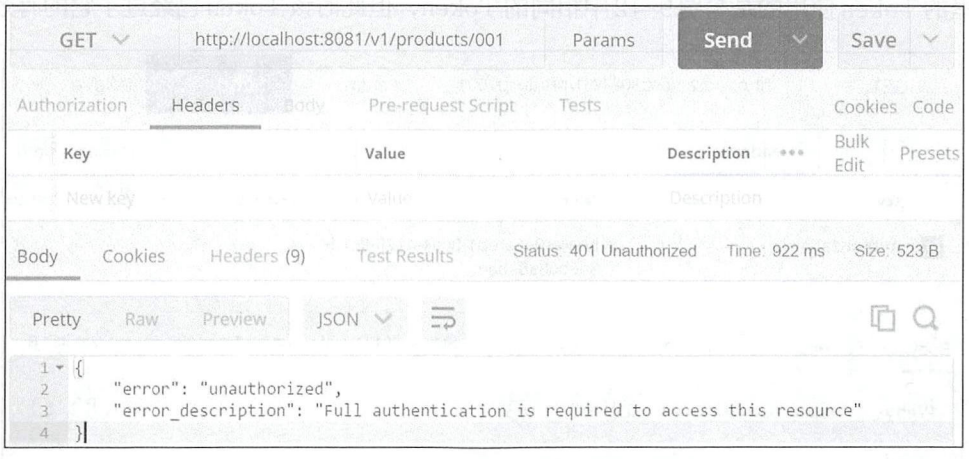


图 9-11　未认证请求下访问资源的结果示意图

现在轮到 Token 发挥作用了，我们在访问图 9-11 中的 http://localhost:8081/v1/products/001 时设置“Authorization”请求头并传入 Token 信息，就会得到如图 9-12 所示的结果。显然，因为我们传入的是有效 Token，所以满足认证要求。

仅供非商业用途或交流学习使用

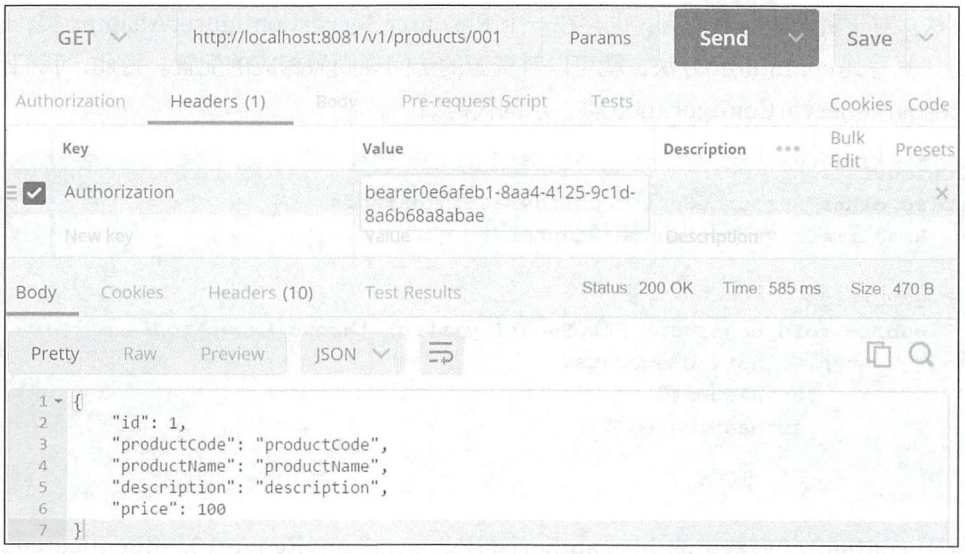


图 9-12　使用有效 Token 访问资源的结果示意图

还有一种情况需要考虑，因为 Token 具有有效期，当使用已过期的 Token 访问 http://localhost:8081/v1/products/001 时，会得到另一种错误信息，如图 9-13 所示。请注意这里的 Token 使用的是与图 9-12 中相同的 Token，但此时该 Token 已经过了它的有效期。

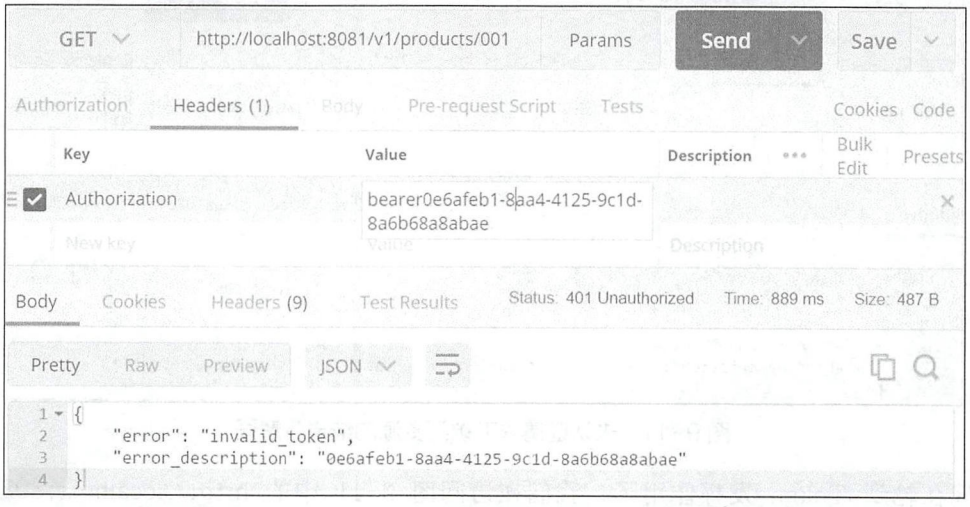


图 9-13　使用无效或已过期 Token 访问资源的结果示意图

当然，如果我们使用的是一个非法的 Token，那显然也会得到图 9-13 所示的“invalid_token”错误。

2. 使用认证用户+角色级别保护服务访问

对于某些安全性要求比较高的资源，通常还需要限定访问资源的角色。例如，相较 Product 服务，我们认为 Order 服务涉及用户下单、支付等核心业务流程，我们不应该开放资源访问入口给所有的认证用户，而是应该限定只有角色为“ADMIN”的管理员才能访问该服务。

使用认证用户+角色保护服务访问的示例代码如下所示。可以看到我们使用 HttpSecurity 类中的 antMatchers("/v1/orders/**")和 hasRole("ADMIN")方法为访问"/v1/orders/**"的请求限定了角色，只有"ADMIN"角色的认证用户才能访问以"/v1/orders/"为根地址的所有 URL。

```
@Configuration
public class ResourceServerConfiguration extends
    ResourceServerConfigurerAdapter{

    @Override
    public void configure(HttpSecurity http) throws Exception {

        http.authorizeRequests()
            .antMatchers("/v1/orders/**")
            .hasRole("ADMIN")
            .anyRequest()
            .authenticated();
    }
}
```

如果使用认证用户+角色的方式来保护服务访问，当我们传递角色为“USER”的认证用户 Token 到 Order 服务时，就会得到如图 9-14 所示的“access_denied”错误信息。

```
{
    "error": "access_denied",
    "error_description": "Access is denied"
}
```

图 9-14　使用 USER 访问 ADMIN 用户限定访问资源的结果示意图

接下来换一个角色来生成 Token。图 9-15 中使用了“tianyalan_admin”这个用户名获取 Token，在 9.2.2 节中我们已经将该用户名对应的角色初始化为“ADMIN”。

在使用图 9-15 中的 Token“491466f-cc5f-49aa-89a8-9ca14e9071b4”访问 http://localhost:8083/v1/orders/1/001 时，将会创建订单并得到图 9-16 中的返回信息。

3. 使用认证用户+角色+操作级别保护服务访问

最后一种保护服务访问的策略粒度最小，在认证用户+角色的基础上再对具体某种 HTTP 操作进行限制。在 Account 服务中，我们认为所有对账户的删除操作都是很危险的，所以可以

使用 http.antMatchers(HttpMethod.DELETE, "/v1/acounts/**")方法来对删除操作进行保护，示例代码如下。

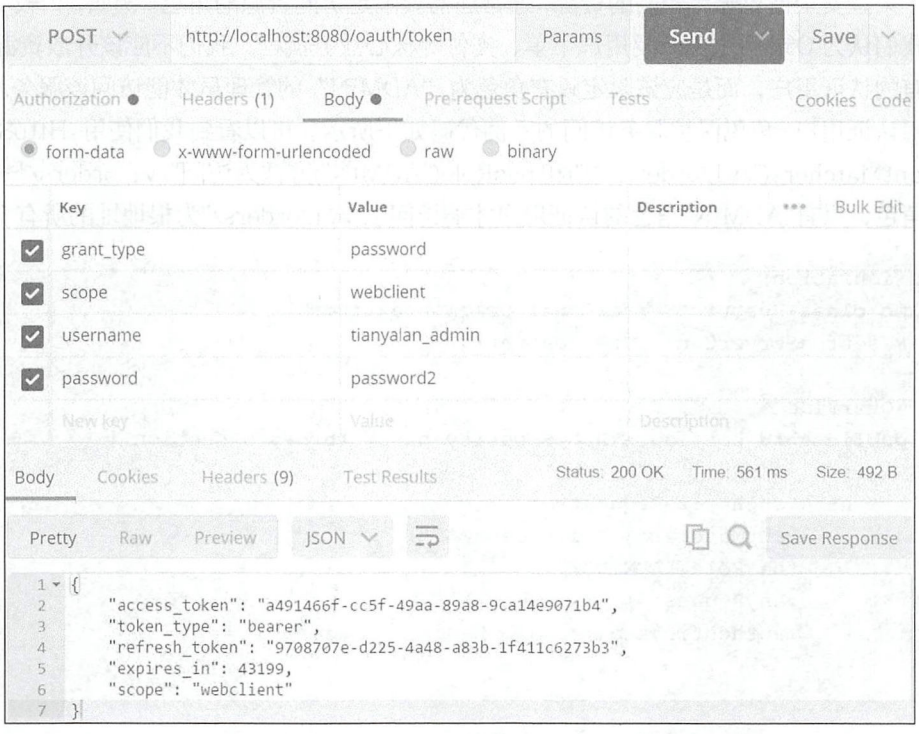


图 9-15　使用 ADMIN 用户生成 Token 的结果示意图

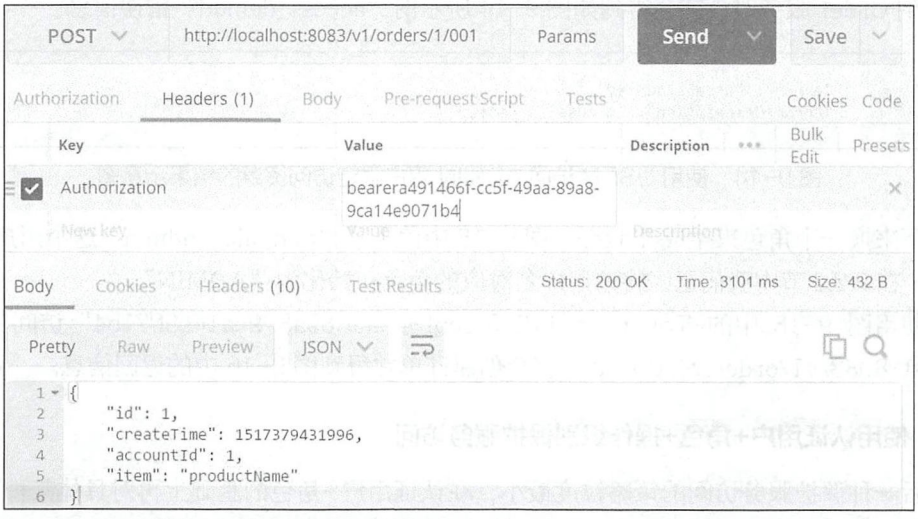


图 9-16　使用由 ADMIN 用户生成 Token 访问资源的结果示意图

```
@Configuration
public class ResourceServerConfiguration extends
    ResourceServerConfigurerAdapter {

    @Override
    public void configure(HttpSecurity http) throws Exception{
        http.authorizeRequests()
            .antMatchers(HttpMethod.DELETE, "/v1/accounts/**")
            .hasRole("ADMIN")
            .anyRequest()
            .authenticated();
    }
}
```

使用“tianyalan_user”生成一个普通用户的认证 Token，然后使用 DELETE 方法访问 http://localhost:8082/v1/accounts/1，得到的还是如图 9-14 所示的“access_denied”错误信息。尝试使用“tianyalan_admin”生成一个管理员用户的认证 Token 并再次访问该地址，就可以得到正常的 200 响应。

9.3.3 使用 OAuth2RestTemplate 传播 Token

截至目前，我们讨论的都是如何对单个服务的访问进行保护，但现实中很多场景都需要服务之间进行交互以完成复杂的业务流程。我们在第 6 章中已经介绍了用于服务之间进行路由和整合的 API 网关模式以及对应的实现工具 Spring Cloud Netflix Zuul。本节将在第 6 章的基础上进一步讨论确保多个服务交互过程安全性的方法和工具。

1. 场景分析

首先来看一个典型的业务场景，在我们的案例系统中存在 Product 服务、Account 服务和 Order 服务这 3 个业务服务，以及用于服务路由的 zuul-server。下面以添加订单流程为例，看一下这个过程中所涉及的服务交互。添加订单的主流程代码如下所示，在添加订单的过程中，首先根据商品编号获取商品信息，然后根据账户 Id 获取用户账户信息，最后再生成订单并持久化到数据库。注意，这里的获取商品和获取用户账户过程分别依赖于 Product 服务和 Account 服务这两个微服务。

```
public Order addOrder(Long accountId, String productCode) {
    Order order = new Order();

    Product product = getProduct(productCode);
    if(product == null) {
```

```
            return order;
        }

        Account account = getAccount(accountId);
        if(account == null) {
            return order;
        }

        order.setAccountId(accountId);
        order.setItem(product.getProductName());
        order.setCreateTime(new Date());

        orderRepository.save(order);

        return order;
    }
```

在服务访问控制上，上述流程需要经历以下几个阶段（见图 9-17）。

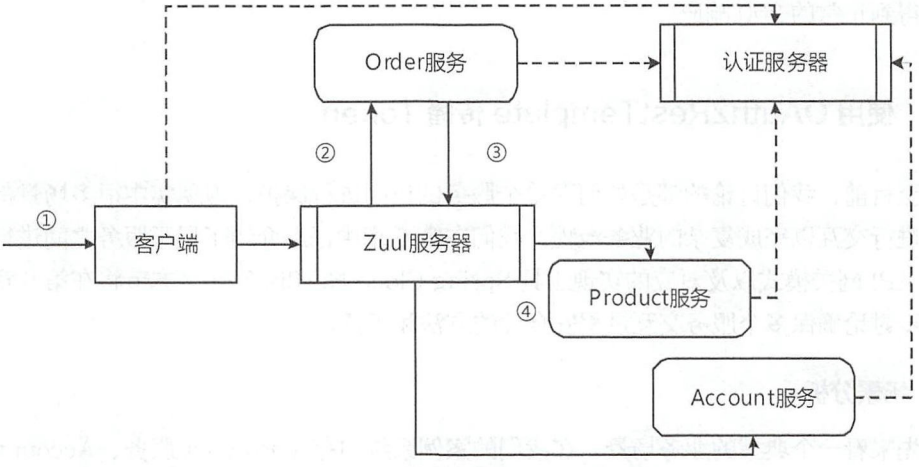


图 9-17　服务调用链与 Token 传播

① 生成 Token。任何想要访问 Order 服务并进行订单创建的客户端都需要首先获取一个有效的 Token。

② 访问 Order 服务。持有 Token 的客户端访问 Order 服务提供的 URL 进行下单操作。请注意，这里同样需要先访问 zuul-server，然后通过 zuul-server 的路由再访问 Order 服务，Order 服务会验证所传入 Token 的有效性。

③ 访问 zuul-server。Order 服务再通过 zuul-server 访问 Product 服务和 Account 服务，这一过程需要确保 Token 在 zuul-server 和 Product 服务、Account 服务之间的有效传播。

④ 访问 Product 服务和 Account 服务。Product 服务和 Account 服务处于流程的末端，

分别对所传入 Token 进行验证并返回相应的结果。

我们在前几章中已经分别完成了图 9-17 中 Zuul 服务器、Product 服务、Account 服务和 Order 服务的构建，但还有一个核心问题没有解决，即如何确保 Token 通过 Zuul 服务器在各个微服务中进行有效地传播。

2. 使用 OAuth2RestTemplate

为了简化在各个服务之间进行 Token 传播的相关操作，Spring Security 提供了 org.springframework.security.oauth2.client.OAuth2RestTemplate 类对 RestTemplate 进行封装。OAuth2RestTemplate 类中的 acquireAccessToken()方法会将从请求中得到的 Token 保存到 OAuth2ClientContext 中，而 OAuth2ClientContext 会把每个用户的请求信息控制在会话范围内，以确保不同用户的状态分离。另一方面，OAuth2RestTemplate 还依赖于 OAuth2ProtectedResourceDetails 类，该类封装了在 9.2.2 节中介绍的客户端 Id、客户端安全码、访问范围等属性。通过传入 OAuth2ClientContext 和 OAuth2ProtectedResourceDetails，我们就可以创建一下 OAuth2RestTemplate 类，示例代码如下所示（位于 OrderApplication 类中）。

```
@Bean
public OAuth2RestTemplate oauth2RestTemplate(
    OAuth2ClientContext oauth2ClientContext,
    OAuth2ProtectedResourceDetails details) {

    return new OAuth2RestTemplate(details, oauth2ClientContext);
}
```

一旦 OAuth2RestTemplate 创建成功，我们就可以使用它来对 Product 服务和 Account 服务进行访问。以下代码位于 order-service 工程的 ProductClient 类中，展示了如何使用 OAuth2RestTemplate 经由 zuul-server 对 Product 服务进行访问的过程，可以看到 OAuth2RestTemplate 类的使用方法与普通的 RestTemplate 完全一致，所有关于 Token 传播的细节已经被完整地封装在每次请求中。

```
@Component
public class ProductClient {
    @Autowired
    OAuth2RestTemplate restTemplate;

    private static final Logger logger =
        LoggerFactory.getLogger(ProductClient.class);

    public Product getProduct(String productCode){
```

```
        logger.debug("In Order Service.getProduct: {}",
            UserContext.getCorrelationId());

        ResponseEntity<Product> restExchange =
            restTemplate.exchange(
        "http://localhost:5555/api/product/v1/products/{productCode}",
            HttpMethod.GET,
            null,
            Product.class,
            productCode);

        Product product = restExchange.getBody();
        return product;
    }
}
```

同样，针对 Account 服务的访问代码位于 AccountClient 类中，采用的是和 ProductClient 完全一样的处理方式，读者可以自行参考。

以下是 Order 服务中创建订单的完整代码（位于 OrderService 类中），通过注入 ProductClient 和 AccountClient 类，整个流程就会将多个微服务整合在一起，并通过 Token 的传播实现服务的安全访问。

```
@Autowired
ProductClient productClient;

@Autowired
AccountClient accountClient;

@HystrixCommand
private Account getAccount(Long accountId) {
    return accountClient.getAccount(accountId);
}

@HystrixCommand
private Product getProduct(String productCode) {
    return productClient.getProduct(productCode);
}

public Order addOrder(Long accountId, String productCode) {
    Order order = new Order();

    Product product = getProduct(productCode);
    if(product == null) {
        return order;
```

```
        }

        Account account = getAccount(accountId);
        if(account == null) {
            return order;
        }

        order.setAccountId(accountId);
        order.setItem(product.getProductName());
        order.setCreateTime(new Date());

        orderRepository.save(order);

        return order;
    }
```

最后，通过 Postman 来验证以上流程的正确性。通过访问 Zuul 中配置的 Order 服务入口 http://localhost:5555/api/order/v1/orders/1/001，并传入角色为“ADMIN”的用户对应的 Token 信息，可以得到如图 9-18 所示的结果，我们看到订单已经被成功创建。请注意，与图 9-12 不同，图 9-18 中的服务请求通过 zuul-server 完成了多个服务之间的交互过程。而在 Product 服务、Account 服务和 Order 服务中，我们可以看到相应的服务调用日志输出。

图 9-18　使用由 ADMIN 用户所生成 Token 访问 zuul-server 创建订单的结果示意图

9.4 本章小结

本章首先介绍了微服务架构中安全性设计的需求和切入点，引出了基于 Token 的认证方案。基于 Token 的认证方案实现上可以有多种方式，本章介绍的是在分布式服务系统中被广泛应用的 OAuth 协议，重点对 OAuth 协议的应用场景、工作流程和授权模式做了全面介绍。

在 OAuth 协议中，实现服务安全访问的首要条件是提供一个认证服务器，通过在认证服务器上初始化用户和客户端信息，就可以获取访问服务所需要的 Token 信息。本章我们借助了 Spring Cloud Security 作为实现认证服务器的工具。

站在单个微服务的角度讲，需要与认证服务器进行集成以便进行 Token 验证，而在服务内部，可以使用不同的访问策略限制服务资源的访问。站在多个服务交互的角度讲，我们需要确保 Token 在各个微服务之间的有效传播。本章对如何实现单个服务以及多个服务交互场景下的安全访问也做了详细介绍。

截至本章，我们已经全面介绍了构建微服务系统的方法和工具。在各个微服务已经构建完成的前提下，下一章的内容将围绕服务监控展开，我们会讨论服务监控的原理和方法以及具体的实现工具 Spring Cloud Sleuth。

第 10 章

Spring Cloud Sleuth 与服务监控

在传统的单块系统中，所有的业务都在同一台服务器上，如果服务运行时出现错误和异常，我们只要关注一台服务器就可以快速定位和处理问题。但在微服务架构中，事情就没有那么简单了。微服务架构在本质上也是一种分布式架构，微服务架构的特点决定了各个服务部署在分布式环境中。各个微服务独立部署和运行，彼此通过网络交互，而且都是无状态的服务，一个客户端请求可能需要经过很多个微服务的处理和传递才能完成业务逻辑。在这种场景下，我们难免会遇到如下两个核心问题：

- 如何管理服务之间的调用关系
- 如何跟踪业务流的处理顺序和结果

本章将从服务监控的角度出发尝试回答上述两个问题，并引出 Spring Cloud 家族中另一个重要成员 Spring Cloud Sleuth，通过工具框架的应用和整合给出服务监控的具体方法和实践。

10.1 服务监控与 Spring Cloud Sleuth

本节将简要分析服务监控的基本原理，这是理解服务监控相关工具和框架的基础。同时，作为 Spring Cloud 中用于实现服务监控的专用工具，Spring Cloud Sleuth 为实现这些基本原理提供了完整而强大的解决方案。

10.1.1 服务监控基本原理

在微服务架构中，我们基于业务划分服务并对外暴露服务访问接口。在中大型系统中，可能需要多个服务相互协同才能完成一个接口的功能，如果链路上任何一个服务出现问题或者网络超时，都会导致接口调用失败。随着业务的不断扩张，服务之间相互调用的关系会越来越复杂。图 10-1 展示了从请求到响应的典型服务调用链路。当用户发起一个请求时，首先到达前端 A 服务，然后分别对 B 服务和 C 服务进行 RPC 调用；B 服务处理完给 A 服务做出响应，但是 C 服务还需要和后端的 D 服务、E 服务及 F 服务交互之后才能返回给 A 服务处理结果，最后由 A 服务来响应用户的请求。

仅供非商业用途或交流学习使用

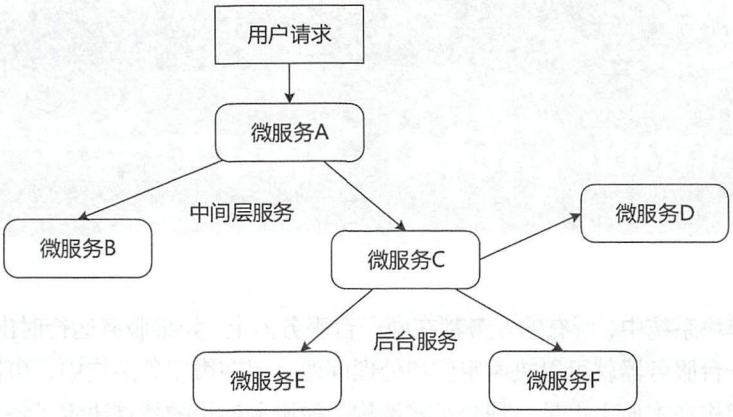


图 10-1　微服务调用链路示意图

服务数量对调用链路分析的复杂度影响较大。在大型微服务系统中，显然通过人工手段已经完全不能在分布式环境下完成这种服务调用链路的分析。为此，我们需要引入分布式服务跟踪机制并借助于一定的工具实现微服务架构下的服务监控。

分布式服务跟踪原理上实际上并不复杂，首先需要引入两个基本概念，即 TraceId 和 SpanId。

- TraceId

TraceId 即跟踪 Id。在微服务架构中，每个请求可生成一个全局的唯一性 Id，通过这个 Id 可以串联起整个调用链，也就是说请求在分布式系统内部流转时，系统需要始终保持传递该唯一性 Id，直到请求返回。这个唯一性 Id 就是 TraceId。

- SpanId

除了 TraceId，我们还需要 SpanId，SpanId 一般被称为跨度 Id。当请求到达各个服务组件时，通过 SpanId 来标识它的开始、具体执行过程和结束。对于每个 Span 而言，它必须有开始和结束两个节点，通过记录开始 Span 和结束 Span 的时间戳统计该 Span 的时间延迟。

整个调用过程中每个请求都要透传 TraceId 和 SpanId。每个服务将该次请求附带的 SpanId 作为 PSpanId（父 SpanId）进行记录，并且生成自己的 SpanId。一个没有 PSpanId 的 Span 即为根 Span，可以看成调用链入口。所以要查看某次完整的调用只需根据 TraceId 查出所有调用记录，然后通过 PSpanId 和 SpanId 组织起整个调用父子关系。

关于 TraceId 和 SpanId 的关系可以通过图 10-2 做进一步解释。在图 10-2 中，我们通过如下 4 种关键事件记录每个服务的客户端请求和服务器响应过程。在如 Spring Cloud Sleuth 这样的工具中，也把这些事件称为注解（Annotation），为了避免混淆“注解”这一专用词汇的应用场景，在本书中我们还是使用“关键事件”来描述监控过程。

- cs 事件

cs 代表 Client Send，客户端发送一个请求，表示 Span 的开始。

仅供非商业用途或交流学习使用

- sr 事件

sr 代表 Server Receive，服务端接收请求并开始处理它。(sr-cs) 值等于请求的网络延迟。

- ss 事件

ss 代表 Server Send，服务端处理请求完成，开始返回结果给客户端。(ss-sr) 值表示服务端处理请求的时间。

- cr 事件

cr 代表 Client Receive，客户端接受返回结果，此时 Span 结束。(cr-sr) 值表示客户端接收服务端数据的时间。

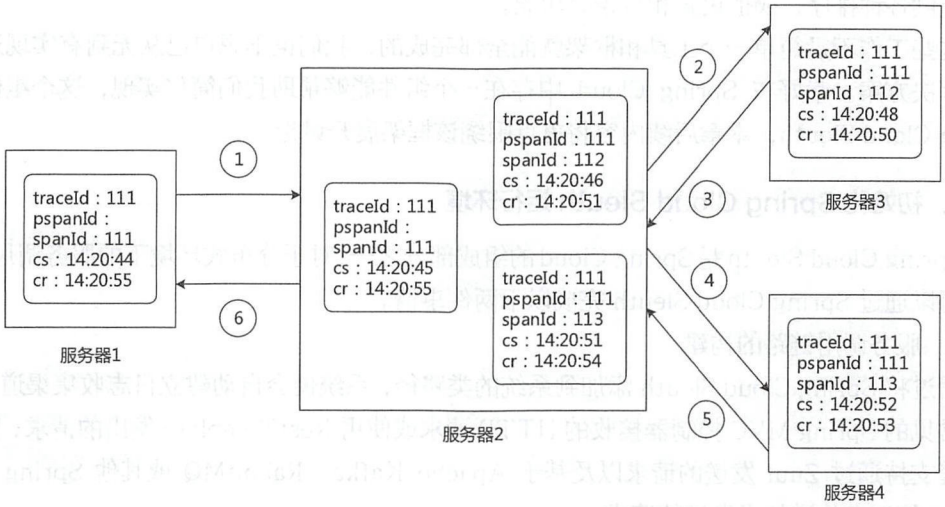


图 10-2　微服务调用链路中 TraceId 和 SpanId 示意图

在图 10-2 中，我们看到这次请求的 TraceId 是 111，而 SpanId 根据不同的服务会发生变化。而 4 种关键事件构成了客户端和服务器对一次请求处理的闭环，对于服务器 1 而言，cs 是 14:20:44，cr 是 14:20:55，也就说该次服务请求经由服务器 1 的时间是 11s (14:20:55-14:20:44)，显然这个响应时间非常长。通过这些关键事件我们就可以发现服务调用链路中存在的问题，目前主流的服务监控实现工具都对这些关键事件做了支持和封装。

10.1.2　引入 Spring Cloud Sleuth

通过上一节的原理分析，我们明确了分布式环境下服务跟踪的载体，即 TraceId 和 SpanId。但实现服务跟踪，我们还需要实现以下需求。

- 调用链数据生成

对整个调用过程的所有应用进行埋点并输出事件。

仅供非商业用途或交流学习使用

- 调用链数据采集

对各个应用中的事件数据进行采集。

- 调用链数据存储及查询

对采集到的数据进行存储，由于事件数据量一般都很大，不仅要能实现存储，还需要能提供快速查询。

- 调用链数据运算

对采集到的事件数据进行各种指标运算，并将运算结果保存起来。

- 调用链数据管理

提供各种排序、阈值设置和警告等功能。

这些工作不是简单一个工具和框架就能全部完成的，我们也不想自己从无到有实现这样一整套解决方案。幸好在 Spring Cloud 中存在一个组件能够帮助我们简化实现，这个组件就是 Spring Cloud Sleuth，本章后续内容将重点围绕该框架展开讨论。

1. 初始化 Spring Cloud Sleuth 运行环境

Spring Cloud Sleuth 是Spring Cloud的组成部分之一，对于分布式环境下的服务调用链路，我们可以通过 Spring Cloud Sleuth 完成以下两件事情。

- 服务调用链路的构建

通过将 Spring Cloud Sleuth 添加到系统的类路径，系统便会自动建立日志收集渠道，不仅包括常见的 Spring MVC 控制器接收的 HTTP 请求或使用 RestTemplate 发出的请求，同时也能无缝支持通过 Zuul 发送的请求以及基于 Apache Kafka、RabbitMQ 或其他 Spring Cloud Stream 等消息传递技术发送的请求。

- 服务监控数据的存储和梳理

针对监控数据的管理，Spring Cloud Sleuth 可以设置常见的日志格式来输出 TraceId 和 SpanId。或者使用诸如 Logstash 等日志发布组件将日志发布到 Elastic Search 等日志分析工具中进行处理。同时，Spring Cloud Sleuth 也兼容了 Zipkin、HTrace 等第三方工具的应用和集成。

借助于 Spring Cloud Sleuth 中简单且自动化的服务调用链路构建过程，想要在某个微服务中添加服务监控功能，要做的事情只有一件，即把 spring-cloud-starter-sleuth 组件添加到 maven 依赖中即可，如下所示。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

为了演示整个服务调用链路，可以根据需要在涉及的各个服务中添加这个 maven 依赖，包

仅供非商业用途或交流学习使用

括 Product 服务、Account 服务、Order 服务以及至关重要的 zuul-server。

2. Spring Cloud Sleuth 中的 Trace 和 Span

初始化工作完成之后，接下来就来看一下引入 Spring Cloud Sleuth 之后为我们带来的变化，首当其冲的切入点是日志分析。

在案例 OrderSystem 中，我们还是通过 Postman 来模拟一次请求操作。在上一章的基础上，通过传入合法的 Token 信息并向 Zuul 中的服务地址 http://localhost:5555/api/order/v1/orders/1/001 发起请求之后，我们来看一下系统中究竟发生了什么。

以 Product 服务为例，在控制台上可以看到如下日志信息。

```
INFO
[productservice,0be0e6c221139b56,72c9b62d91eb08fa,true]
14168 --- [nio-8081-exec-1] o.s.b.a.s.o.r.UserInfoTokenServices:
Getting user info from: http://localhost:8080/user
```

显然，对 Product 服务的请求需要与认证服务器进行 Token 认证，所以会有上面这条日志记录。请注意斜体部分的日志，包括以下四段内容。

- 服务名称

第一段中的 productservice 代表该服务的名称，使用的就是在 bootstrap.yml 中 spring.application.name 指定的服务名称。考虑到服务跟踪的需求，为服务指定一个统一且友好的名称是一项最佳实践。

- TraceId

TraceId 代表一次完整请求的唯一编号，上例中的 0be0e6c221139b56 就是该次请求的唯一编号。在诸如 Zipkin 等可视化工具中，可以通过 TraceId 查看完整的服务调用链路。

- SpanId

在一个完整的服务调用链路中，每一个服务之间的调用过程都可以通过 SpanId 进行唯一标识（如上例中的 72c9b62d91eb08fa），所以 TraceId 和 SpanId 是一对多的关系，即一个 TraceId 一般都会包含多个 SpanId，每一个 SpanId 都从属于特定的 TraceId。同样，也可以通过 SpanId 查看某一个服务调用过程的详细信息。

- Zipkin 标志位

该标志位用于识别是否将服务跟踪信息同步到 Zipkin，Zipkin 是一个可视化工具，可以将服务跟踪信息通过一定的形式展示出来。因为在示例运行过程中，我们已经启动了 Zipkin 服务器，所以上例中的标志位值为 true，意味着所有跟踪信息将被同步到 Zipkin 中。

在引入 Spring Cloud Sleuth 之前，以上四段内容的日志显示效果为[productservice,,,]，可以看到 TraceId、SpanId 和 Zipkin 标志位都为空，这些内容都是在引入 Spring Cloud Sleuth 相关 jar 包之后自动被添加到了每一次服务调用中。

仅供非商业用途或交流学习使用

下面再来看一下 Account 服务和 Order 服务中的日志信息。Account 服务中的控制台输出日志如下，可以看到用斜体表示的完整四段内容，其中，TraceId 为 0be0e6c221139b56，SpanId 为 4dbc9a362472cb01，Zipkin 标志位为 true。

```
2018-03-30 15:54:19.063 INFO
[accountservice,0be0e6c221139b56,4dbc9a362472cb01,true]
13116 --- [nio-8082-exec-1] o.s.b.a.s.o.r.UserInfoTokenServices:
Getting user info from: http://localhost:8080/user
```

同样，在 Order 服务中的日志信息中，我们同样也发现了类似的一条记录，其中，TraceId 为 0be0e6c221139b56，SpanId 为 992aec60c399ece2，Zipkin 标志位也为 true。

```
2018-03-30 15:54:19.080 INFO
[orderservice,0be0e6c221139b56,992aec60c399ece2,true]
3888 --- [nio-8083-exec-4] o.s.b.a.s.o.r.UserInfoTokenServices:
Getting user info from: http://localhost:8080/user
```

请注意，以上三段日志中的 TraceId 都是 0be0e6c221139b56，也就是它们属于同一个服务调用链路，而不同的 SpanId 代表着整个链路中的具体某一个服务调用。从日志中的时间上也不难看出三者之间的调用时序。基于这 3 个服务以及 TraceId、SpanId 所生成的服务调用时序链路效果如图 10-3 所示。关于该链路的可视化效果和更详细的数据信息将在下一节中进行具体展开。

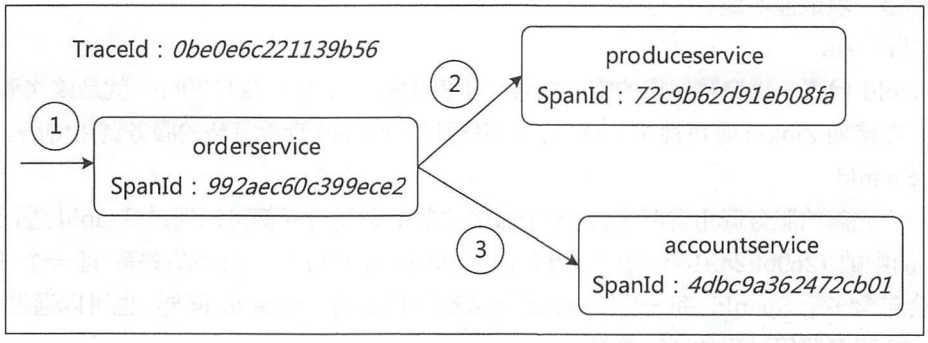


图 10-3　3 个服务调用链路效果图

3. 使用 Tracer 接口

除了在日志中捕获跟踪信息，还可以使用 Spring Cloud Sleuth 提供的工具类在代码级别控制跟踪信息的生成和展示。通过 org.springframework.cloud.sleuth.Tracer 接口所提供的 API 可以观察到 Span 的各个生命周期。Tracer 接口常见的方法和功能如下所示。

- createSpan()方法和 closeSpan()方法

以下代码展示了如何手工创建一个 Span 实例，如果当前线程上下文中已经存在一个 Span

仅供非商业用途或交流学习使用

了，那已存在的 Span 就会成为新创建 Span 的父级。通过 tracer.createSpan()方法创建新的 Span 时，可以给该 Span 打 Tag 并在 Span 上记录事件，如果想要把该 Span 信息发送给 Zipkin，就不要忘记使用 closeSpan()方法关闭 Span。tracer.createSpan()还有一个重载方法可以显式指定父 Span。

```
Span newSpan = tracer.createSpan("newSpan");
    try {
        // 给一个 Span 打 Tag
        this.tracer.addTag("tagName", tagValue);

        // 在 Span 上记录事件
        newSpan.logEvent("eventOccured");
    } finally {
        //关闭 Span
        this.tracer.close(newSpan);
}
```

- getCurrentSpan()方法

使用 getCurrentSpan()方法可以获取当前线程下的 Span 信息，这个方法非常实用，因为多数情况下我们的主要目标是获取服务链路上的各个 Trace 和 Span 信息。例如，在某一个服务链路中，想要获取该链路的 TraceId，可以通过以下语句实现这一目标。

```
tracer.getCurrentSpan().traceIdString()
```

当通过 tracer.getCurrentSpan()方法获取特定 Span 时，还可以使用 getSpanId()、getTraceId()、getName()、getParents()等方法获取更多有用的信息。

Tracer 接口的应用场景非常广泛，一个典型的应用场景就是在 HTTP 头中添加 TraceId 或 SpanId 以实现对每个请求进行分析和管理。根据 Tracer 接口并结合第 6 章中介绍的 ZuulFilter 机制，可以简单实现这一功能，示例代码如下。

```
@Component
public class TraceFilter extends ZuulFilter{

    @Autowired
    Tracer tracer;

    @Override
    public String filterType() {
        return "post";
    }

    @Override
```

仅供非商业用途或交流学习使用


```
    public int filterOrder() {
        return 1;
    }

    @Override
    public boolean shouldFilter() {
        return true;
    }

    @Override
    public Object run() {
        RequestContext ctx = RequestContext.getCurrentContext();

        ctx.getResponse().addHeader("trace-id",
            tracer.getCurrentSpan().traceIdString());

        return null;
    }
}
```

上述代码首先通过 tracer.getCurrentSpan()方法获取当前 Span，然后再通过该 Span 获取本次请求对应的 TraceId 信息。现在，访问 Zuul 中的相关服务时，每一个请求中就会带有这个 HTTP 头。例如，当我们通过 Postman 访问 http://localhost:5555/api/order/v1/orders/1/001 端点时，相应的 HTTP 头列表中就多了 trace-id 这一项（见图 10-4）。我们也可以根据需要添加 SpanId 或其他所需的任何信息。

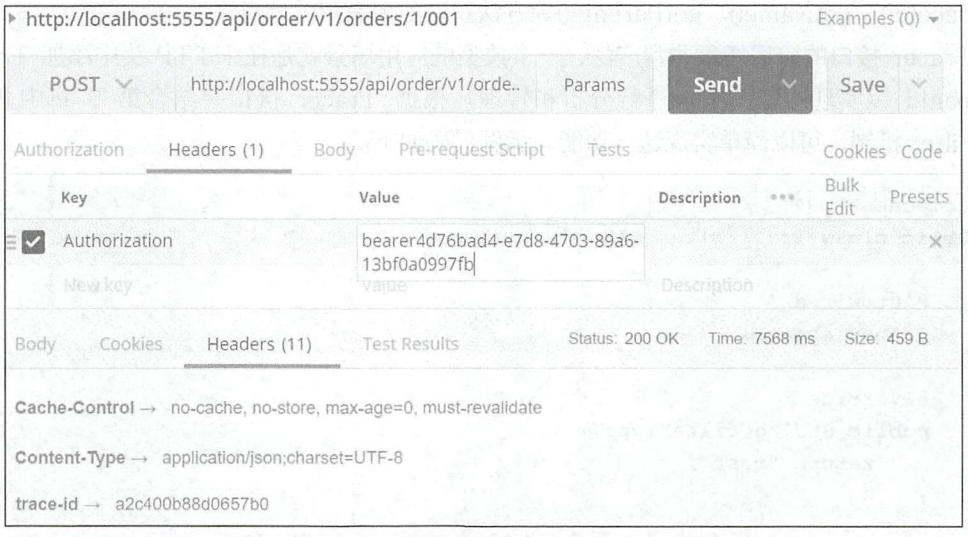


图 10-4　在 HTTP 头中添加 TraceId 效果图

仅供非商业用途或交流学习使用

使用 Spring Cloud Sleuth 可以实现强大的服务调用跟踪功能，但是从定位上讲，Spring Cloud Sleuth 并不是一个完整的解决方案。在接下去的内容中，我们将通过 Zipkin 这一可视化工具丰富 Spring Cloud Sleuth 的使用体验，而 Spring Cloud Sleuth 在设计上也充分考虑到了与其他第三方工具进行整合和集成的需要。

10.2 整合 Spring Cloud Sleuth 与 Zipkin

Spring Cloud Sleuth 的强大之处并不是体现在独立的服务跟踪和日志处理能力上，而是体现在框架整合能力上。业界关于日志聚合(Log Aggregation)的主流实现方案包括 ELK(Elastic Search、Logstash 和 Kibana) 等，Spring Cloud Sleuth 可以整合这些日志聚合方案。虽然以 ELK 为代表的日志聚合工具提供的收集、存储和检索功能已十分强大，但是 ELK 平台上的数据分析维度缺少对请求链路中各阶段时间延迟的关注。对于如何分析一个请求链路中出现延迟过高的瓶颈问题，像 ELK 这样的日志聚合系统就无法满足需求。Spring Cloud Sleuth 在框架整合的基础上可以方便地引入 Twitter Zipkin 等工具实现这一难题。

本节首先分析 Zipkin 的基本结构，然后介绍构建 Zipkin 服务器的方法，最后详细阐述使用 Zipkin 实现服务可视化跟踪的具体方法。

10.2.1 Zipkin 基本结构

Zipkin 整体架构如图 10-5 所示，包括 4 个主要组件。

- Collector

收集器组件，接收来自各个服务传输的数据，将这些数据转换为 Zipkin 内部处理的 Span 格式，兼顾数据收集和格式化的功能。

- Storage

存储组件，存储收集而来的数据，当前支持 Cassandra、Redis、HBase、MySQL、PostgreSQL、SQLite 等工具，默认存储在内存中。

- RESTful API

API 组件，负责查询 Storage 中存储的数据，提供简单的 RESTful API 获取数据，主要提供给 Web UI 使用。

- Web UI

UI 组件，提供简单的 Web 界面，基于 API 组件的上层应用，可以方便、直观地查询和分析跟踪信息。

仅供非商业用途或交流学习使用

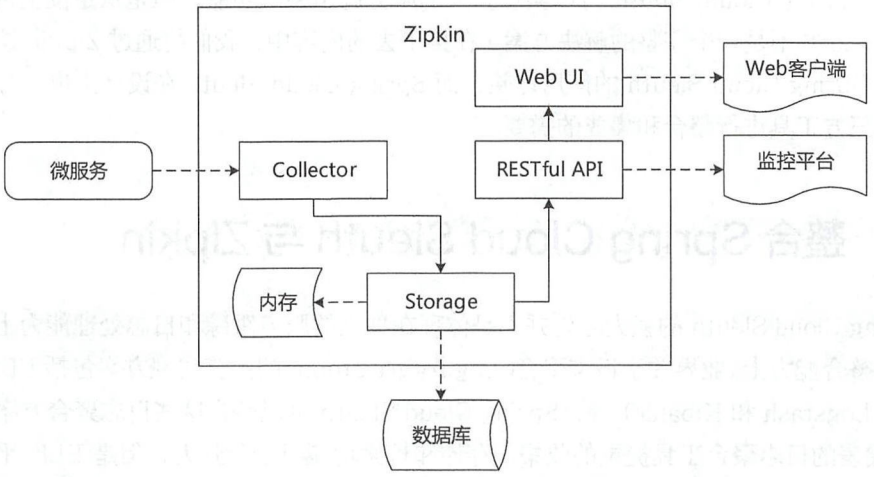


图 10-5　Zipkin 基本架构

在运行过程中，可以通过 Zipkin 获取类似图 10-6 所示的服务调用链路分析效果，从图中可以看到，Zipkin 为用户提供了强大的可视化管理功能。

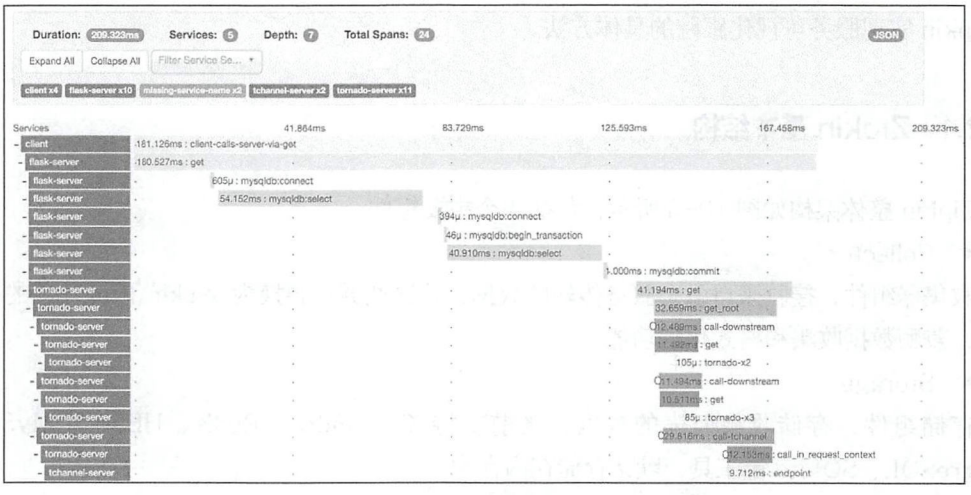


图 10-6　Zipkin 服务调用链路分析示例图

10.2.2　引入 Zipkin

在 Spring Cloud Sleuth 中整合 Zipkin 也非常简单，通过搭建 Zipkin 服务器并为各个微服务集成 Zipkin 服务即可完成准备工作。

1. 构建 Zipkin 服务器

构建 Zipkin 服务器的第一步是创建一个新的 zipkin-server 工程并添加 Zipkin 相关的 maven 依赖，包括 Zipkin 服务器和 UI 组件，如下所示。

```
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-server</artifactId>
</dependency>

<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-autoconfigure-ui</artifactId>
</dependency>
```

添加完所需的依赖之后，就可以通过@EnableZipkinServer 注解来构建 Bootstrap 类，代码如下所示。

```
@SpringBootApplication
@EnableZipkinServer
public class ZipkinServerApplication {
    public static void main(String[] args) {

        SpringApplication.run(ZipkinServerApplication.class, args);
    }
}
```

上面提到 Zipkin 的存储组件支持多种实现方案，为了简便起见，本书使用默认的内存存储来演示 Zipkin 的核心功能。用户如果需要支持某些特定的存储组件就应该添加对应的 maven 依赖，例如使用 Mysql 存储组件就需要添加如下依赖。

```
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-autoconfigure-storage-mysql</artifactId>
</dependency>
```

同时，在 application.yml 配置文件中添加类似如下的配置项即可完成 Mysql 存储组件的集成，配置内容如下。

```
server:
  port: 9411

zipkin:
  storage:
```

```
    type: mysql

spring:
  datasource:
    schema: classpath:/mysql.sql
    url:jdbc:mysql:// 127.0.0.1:3306/zipkin
    username: tianlayan
    password: default_pwd
```

2. 集成 Zipkin 服务器

在各个微服务中，需要确保添加了对 Spring Cloud Sleuth 和 Zipkin 的 maven 依赖，如下所示。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
```

然后，在配置文件中添加对 Zipkin 服务器的引用即可，配置内容如下所示。

```
spring:
    zipkin:
        baseUrl: http://localhost:9411
```

至此，Zipkin 环境已经搭建完毕，我们可以通过访问 http://localhost:9411 来获取 Zipkin 所提供的所有可视化结果，接下来将演示如何使用 Zipkin 跟踪服务调用链路。

10.2.3 使用 Zipkin 跟踪服务调用链路

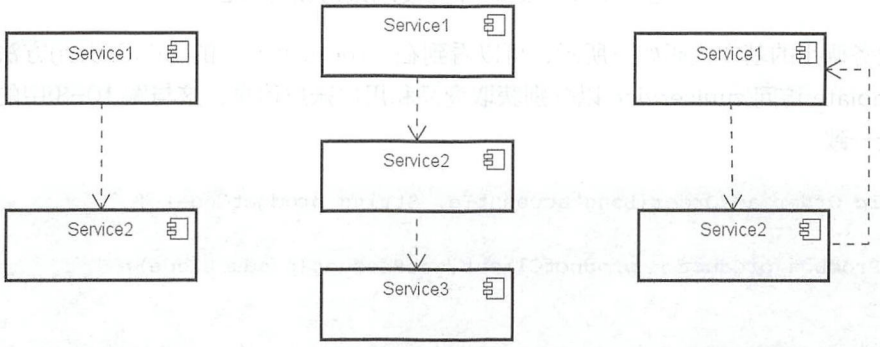
本节将分别从可视化服务依赖关系、可视化服务调用时序和获取服务调用数据这 3 个维度介绍 Zipkin 的强大功能。

1. 可视化服务依赖关系

依赖在某种程度上不可避免，但是过多的依赖势必会增加系统复杂性和降低代码维护性，从而成为团队开发的一种阻碍。在微服务架构中存在很多服务，服务需要管理相互之间的依赖

关系。

依赖关系有3种基本的表现形式，其中，类似Service1依赖于Service2这种直接依赖最容易识别和管理(见图10-7(a));间接依赖即直接依赖关系的衍生,当Service1依赖Service2,而Service2又依赖Service3时,Service1就与Service3产生了间接依赖关系(见图10-7(b));所谓循环依赖就是Service1和Service2之间相互依赖(见图10-7(c))。循环依赖有时候并不像图中描述的那么容易识别，因为产生循环依赖的多个组件之间可能同时存在各种直接或间接依赖关系。



(a) 直接依赖　(b) 间接依赖　(b) 循环依赖

图10-7　服务依赖的3种基本关系

在微服务架构中，微服务之间可能会存在循环依赖从而形成一个网状关系。原则上，在一个设计合理的系统架构中，服务之间不应该存在循环依赖。当然，这里的服务指的不是微服务架构中的所有服务，而是指业务服务。因为对于作为API网关组件的Zuul而言，所有的请求会首先通过Zuul进行访问，然后Zuul作为协调者又会对服务请求进行路由和转发，所以会形成一种环状的调用效果。

当系统规模越来越大后,各个业务服务之间的直接依赖和间接依赖关系就会变得十分复杂。我们需要通过一个简洁明了的可视化工具来查看当前服务链路中的依赖关系，Zipkin就提供了这方面的支持。

当我们通过Postman访问http://localhost:5555/api/order/v1/orders/1/001时,图10-8展示了通过Zipkin获取的服务调用依赖关系。在这个服务调用链路中，我们首先通过zuulservice访问orderservice，然后orderservice又通过zuulservice分别访问productservice和accountservice，从而形成一次完整的业务调用。为了上下文描述的一致性，本章后续内容中将使用zipkin中所显示的zuulservice、orderservice、productservice、accountservice来分别代表Zuul服务、Order服务、Product服务和Account服务。

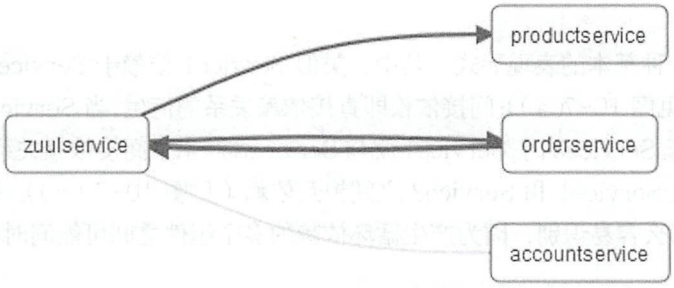


图 10-8　Zipkin 中的服务依赖关系示意图

该业务调用的基本代码如下所示，可以看到在 orderservice 的 addOrder()方法中会通过 RestTemplate 访问 zuulservice 以分别获取商品和用户账户信息，这与图 10-8 中的服务依赖关系完全一致。

```
public Order addOrder(Long accountId, String productCode) {

    Product product = productClient.getProduct(productCode);
    …

    Account account = accountClient.getAccount(accountId);;
    …
}

@Component
public class ProductClient {

    public Product getProduct(String productCode){
        ResponseEntity<Product> restExchange =
            restTemplate.exchange(
        "http://localhost:5555/api/product/v1/products/{productCode}",
        HttpMethod.GET, null, Product.class, productCode);
    }
}
```

在 Zipkin 中同样提供了针对某一个服务的依赖关系视图，通过该视图可以获取单个服务的入口和出口服务。例如，图 10-9 展示的就是 zuulservice 的服务依赖关系视图。对于服务调用链路而言，zuulservice 是处于核心位置的 API 网关服务，从图 10-9 中可以看到它被 orderservice 所使用，同时调用了 accountservice、orderservice 和 productservice 等 3 个不同的业务服务。

Zipkin 为我们管理服务依赖关系提供了一种有效手段，如果我们通过 Zipkin 发现业务服务之间存在循环依赖，就可以采用一定的措施来消除循环依赖。消除循环依赖的主要方式有 3 种，

分别是上移、下移和回调，关于这3种方式的详细描述可以参考笔者的《系统架构设计》一书。此外，系统中存在过于复杂的间接依赖也是架构腐化的一种表现形式，通常可以通过《微服务设计原理与架构》中提到的共享库、共享服务、代码转移、代码冗余、提取新服务和重写服务等方式来优化现有架构。

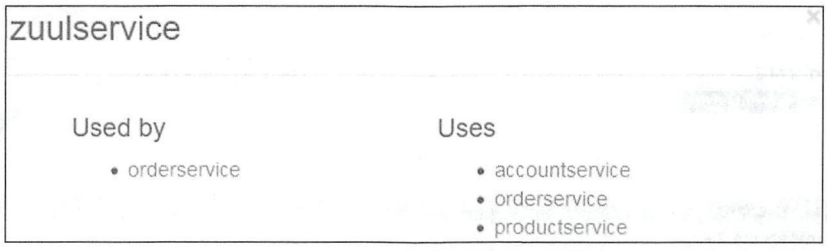


图 10-9　zuulservice 的服务依赖关系图

2. 可视化服务调用时序

可视化服务调用时序是 Zipkin 最重要的功能，对于服务监控而言，服务调用链数据收集、分析和管理的目的是为了发现服务调用过程的问题并采取相应的优化措施。

图 10-10 展示了 Zipkin 可视化服务调用时序的主界面，该主界面主体是一个面向查询的操作界面，其中我们需要关注以下几项内容。

- 服务名称和端点

服务调用链路中的所有服务都会出现在“服务名称”列表中，同时，针对每个服务，我们也可以选择自身感兴趣的端点信息。

- 查询过滤器

Zipkin 支持的查询过滤器包括时间、调用时长（Duration）、关键词以及各项排序功能。

- 查询结果

针对某个服务，Zipkin 的查询结果展示了包含该服务的所有调用链路。如在图 10-10 中，所选的 accountservice 出现在两个服务调用链路中。这两个服务调用实际上是同一个请求（即上文中的 http://localhost:5555/api/order/v1/orders/1/001），但是可以看到这两个服务调用本身以及 accountservice 在这两个服务调用链路中的监控结果具有一定的差别，主要体现在服务调用时长在整个链路上的占比（一个是 2%，一个是 40%）。图 10-10 中 Zipkin 可视化的显示效果也直观地反映了这一差异。

点击图 10-10 中的某一个服务调用链路会得到如图 10-11 所示的服务调用链路明细图，该图展示了几项重要的监控信息。

- Span 信息

一个服务调用链路被分解成若干个 Span，每个 Span 代表完整调用链路中的一个可以衡量的部分。在图 10-11 中，我们看到该调用链路中存在 6 个 Span，每个 Span 的时延都已经被量

仅供非商业用途或交流学习使用

化，并通过背景颜色的深浅来表示时延的大小。

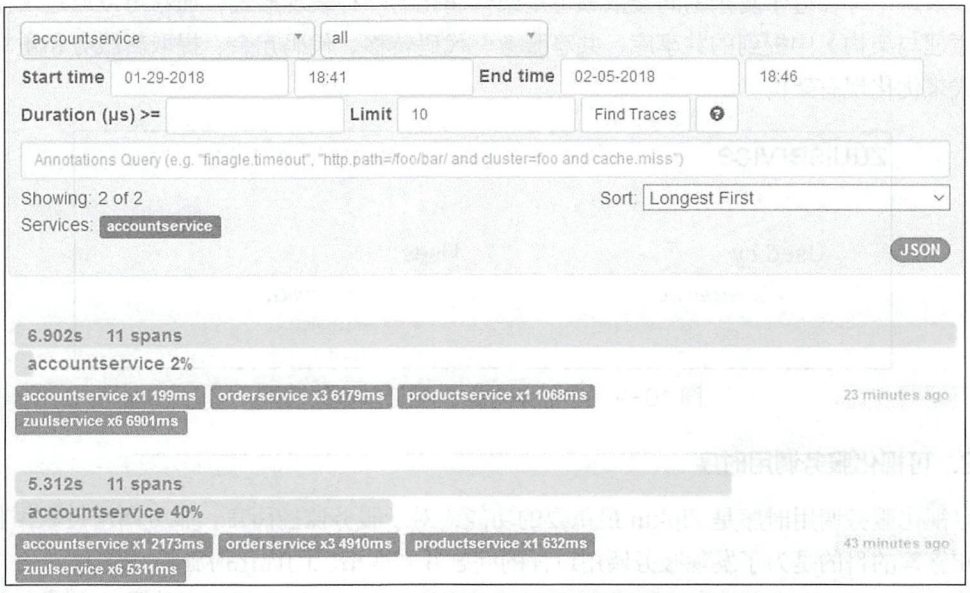


图 10-10　Zipkin 可视化服务调用时序的主界面

- 时序信息

图 10-11 中另外一个重要信息在于各个服务的调用时序，可以看到这 6 个 Span 一共代表 4 个服务，其中，zuulservice 是整个链路的入口服务并且出现了 3 次。通过观察服务之间的调用时序，可以在图 10-8 和图 10-9 的服务依赖关系的基础上给出更为明确的服务调用关系。

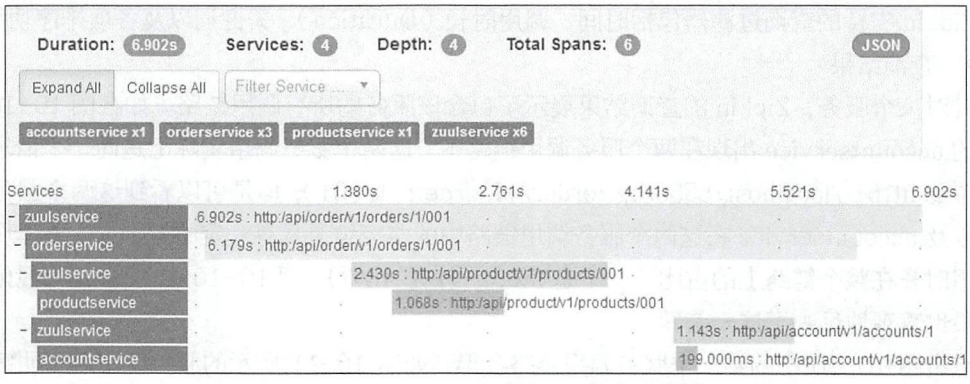


图 10-11　Zipkin 服务调用链路明细图界面

3. 获取服务调用数据

在图 10-11 中，点击任何一个感兴趣的 Span 就可以获取该 Span 对应的各项服务调用数

仅供非商业用途或交流学习使用

据明细。例如，点击 orderservice 中 http:// api/order/v1/orders/1/001 端点对应的 Span，Zipkin 会跳转到一个新的页面并显示如图 10-12 所示的数据。

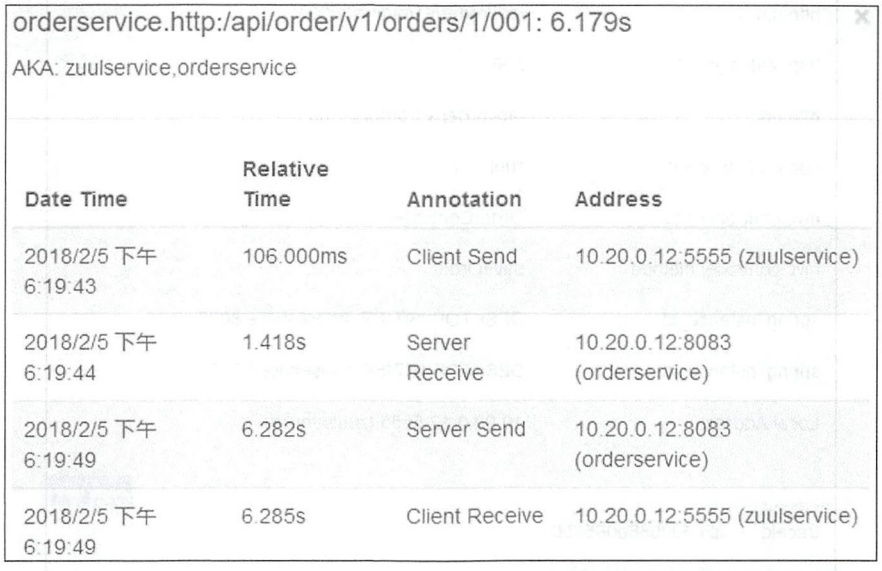


图 10-12　Zipkin 中 Span 对应的 4 个关键事件数据界面

从图 10-12 中可以看到针对该 Span 的 cs、sr、ss 和 cr 这四个关键事件数据。对于 orderservice 而言，zuulservice 相当于服务的调用方，所以服务调用从 zuulservice 发起，通过 1.312s（1.418-0.106）的网络传输时间到达 orderservice，而 orderservice 处理该请求的时间为 4.864s（6.282-1.418），最后请求结果返回给 zuulservice。从这些关键事件的数据中可以得出一个结论，即该请求的整个服务响应时间主要取决于 orderservice 自身的处理时间。

点击图 10-12 中任何一个关键事件，可以得到如图 10-13 所示的明细数据，其中既包括 HTTP 请求相关的方法、路径、状态码等各项基础数据，又包括使用 Spring 构建 RESTful 风格调用时的 Controller 类以及端点信息，还包括服务调用方和服务提供方各自的运行时实例。

在图 10-13 中点击“More Info”按钮时，还可以得到本次调用中用于监控的最重要的元数据，即 TraceId 和 SpanId，如图所示分别为 a2c400b88d0657b0 和 0ca498c65f43d348。注意图 10-13 中还列出了 ParentId，这里的 ParentId 和 TraceId 具有相同值说明本次调用是整个服务调用链路的入口。在其他调用中，ParentId 值不一定和 TraceId 值相同，读者可以自行进行尝试和验证。

最后，作为数据管理和展示的统一平台，Zipkin 还实现了更为低层的数据表现形式，也就是通过 JSON 数据提供对调用过程的详细描述。如下所示的就是一份示例 JSON 数据，描述的也是关键事件所应该具备的各项信息。

仅供非商业用途或交流学习使用

Key	Value
http.method	POST
http.path	/api/order/v1/orders/1/001
http.status_code	200
http.url	/api/order/v1/orders/1/001
Local Component	zuul
mvc.controller.class	OrderController
mvc.controller.method	saveOrder
spring.instance_id	DESKTOP-IBI7JFP:orderservice:8083
spring.instance_id	DESKTOP-IBI7JFP:zuulservice:5555
Local Address	10.20.0.12:5555 (zuulservice)

More Info

traceId	a2c400b88d0657b0
spanId	0ca498c65f43d348
parentId	a2c400b88d0657b0

图 10-13　Zipkin 中每个关键事件明细数据界面

```
[
  {
    "traceId": "3738a813d0c5eb2a",
    "id": "3738a813d0c5eb2a",
    "name": "http:/api/order/v1/orders/1/001",
    "timestamp": 1522401374306000,
    "duration": 5325440,
    "annotations": [
      {
        "timestamp": 1522401374306000,
        "value": "sr",
        "endpoint": {
          "serviceName": "zuulservice",
          "ipv4": "10.20.0.12",
          "port": 5555
        }
      },
      {
        "timestamp": 1522401379630000,
```

仅供非商业用途或交流学习使用

```
      "value": "ss",
      "endpoint": {
        "serviceName": "zuulservice",
        "ipv4": "10.20.0.12",
        "port": 5555
      }
    }
  ],
  "binaryAnnotations": [
    {
      "key": "http.status_code",
      "value": "500",
      "endpoint": {
        "serviceName": "zuulservice",
        "ipv4": "10.20.0.12",
        "port": 5555
      }
    },
    {
      "key": "mvc.controller.class",
      "value": "BasicErrorController",
      "endpoint": {
        "serviceName": "zuulservice",
        "ipv4": "10.20.0.12",
        "port": 5555
      }
    },
    {
      "key": "mvc.controller.method",
      "value": "error",
      "endpoint": {
        "serviceName": "zuulservice",
        "ipv4": "10.20.0.12",
        "port": 5555
      }
    },
    {
      "key": "spring.instance_id",
      "value": "DESKTOP-IBI7JFP:zuulservice:5555",
      "endpoint": {
        "serviceName": "zuulservice",
        "ipv4": "10.20.0.12",
        "port": 5555
      }
    }
```

仅供非商业用途或交流学习使用

```
        ]
    },
    ... 省略其他 Span 信息
]
```

事实上，Zikpin 还提供了专门的 RESTful API 获取一次服务调用链路中所有 Span 对应的 JSON 数据，如本例中的 SpanId 为 3738a813d0c5eb2a，所以对应的 API 即为 http://localhost:9411/api/v1/trace/3738a813d0c5eb2a，我们可以根据需要获取所需的各项信息。

10.2.4 使用 Zipkin 实现自定义跟踪

在前面的介绍中，我们都是基于几个微服务之间的调用关系来讨论 Zipkin 在服务监控可视化过程中发挥的作用，其中完整服务调用链路中的各个 Span 都是采用默认的服务调用结果。在多种情况下，我们通过这些 Span 就可以分析和排查服务调用链路中可能存在的问题。但在某些特定场景下，我们希望在这些 Span 的基础上能够实现一些定制化的数据收集和展示方式。

我们来考虑如下场景，在图 10-11 中我们发现 orderservice 处理该请求的时间为 4.864s，显然这个响应时间是比较长的，但简单通过一个 Span 通常无法判断响应时间过长的原因。那么就可能出现一个需求，即通过添加自定义 Span 的方式进一步对长时间的服务调用进行拆分，把该请求中所涉及的多种操作分别创建 Span，这也是服务监控系统实现过程中的一项最佳实践。

1. 使用 Tracer 添加自定义 Span

使用 Spring Cloud Sleuth 实现自定义 Span 的过程主要使用 Tracer 接口，关于 Tracer 接口已在 10.1.2 节中做了详细介绍。我们知道可以通过 Tracer 接口创建 Span 并把该 Span 的相关信息推送给 Zipkin。现在，我们希望在 productservice 的调用过程中添加一个新的 Span 以帮助 productservice 诊断响应时间过长问题，示例代码如下所示。

```
@Service
public class ProductService {

    @Autowired
    private ProductRepository productRepository;

    @Autowired
    private Tracer tracer;
```

仅供非商业用途或交流学习使用

```
    public Product getProductByCode(String productCode) {
        Span newSpan = tracer.createSpan("getProductByCode");

        try {
            return productRepository.findByProductCode(productCode);
        }
        finally{
          newSpan.tag("peer.service", "database");
          newSpan.logEvent(Span.CLIENT_RECV);
          tracer.close(newSpan);
        }
    }
}
```

上述示例通过以下几个简单的方法调用就可以实现一个自定义 Span。

- tracer.createSpan()方法

该方法用于创建一个自定义的 Span，并将该 Span 命名为“getProductByCode”。

- span.tag()方法

该方法为 Span 打上一个标签，其中两个参数分别代表标签的 Key 和 Value，上述示例中我们创建了一个“peer.service”标签，并把标签值设置为“database”表示该标签与数据库操作相关。

- span.logEvent()方法

该方法用于记录事件，上述示例中记录的是 cr 事件。

- tracer.close()方法

该方法用于关闭 Span，如果不调用该方法就说明 Span 一直处于打开状态。一般在具体操作结束之后必须调用此方法，否则 Span 数据不会被发送到 Zipkin 中。

2. 可视化自定义 Span

重新启动 produceservice 后，在 Postman 中访问 http://localhost:5555/api/order/v1/orders/1/001，就会得到如图 10-14 所示的界面。

在图 10-14 中，可以看到在原有默认可视化效果的基础上又多了一个名为“getProductByCode”的自定义 Span。点击该 Span，将得到如图 10-15 所示的关键事件明细数据界面。

基于图 10-14 和图 10-15 中的数据，我们发现在 orderservice 处理请求的时间中有 19.494ms 是消耗在 productservice 访问数据库以获取商品数据的过程中。同样，我们也可以在 accoutservice 中添加不同的 Span 以实现对服务调用过程更加精细化的管理。

仅供非商业用途或交流学习使用

图 10-14　Zipkin 中添加自定义 Span 的界面

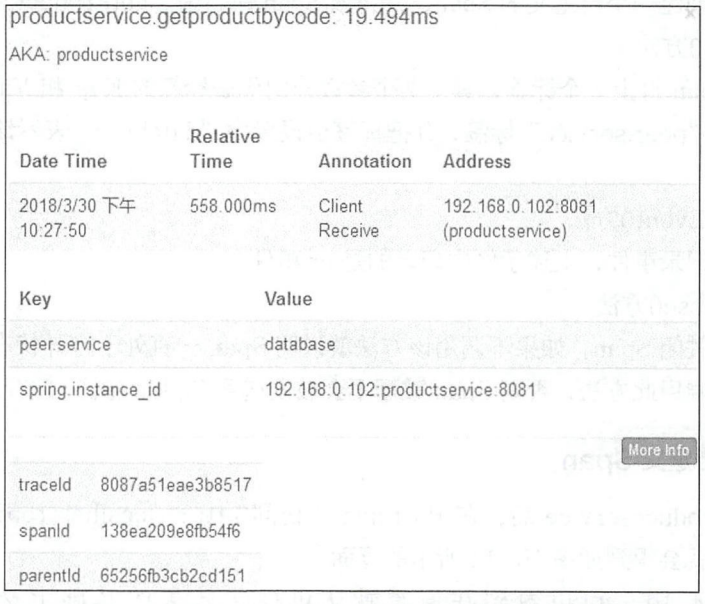


图 10-15　Zipkin 中自定义 Span 中每个关键事件明细数据界面

10.3　本章小结

在微服务架构中，我们需要对当前系统中的服务运行状态进行跟踪和监控，在 Spring Cloud 中同样提供了专门用于分布式服务跟踪的 Spring Cloud Sleuth 组件。

仅供非商业用途或交流学习使用

本章首先介绍分布式环境下服务监控的基本原理，服务监控的基本原理其实并不复杂，业界早已形成了一套比较统一的专业术语和方法论。Spring Cloud Sleuth 就是基于这些原理的一种具体实现方式。Spring Cloud Sleuth 为我们提供了一种代码免侵入的整合方案，在多个服务进行交互的过程中，通过获取 Trace 和 Span 信息就能构建整个服务调用链路。同时，Spring Cloud Sleuth 还提供了 Tracer 接口供开发人员创建灵活而强大的监控方案。

Spring Cloud Sleuth 的另一个重要特性体现在框架整合能力上，在本章中，我们也基于 Twitter 的 Zipkin 讨论了集成方案。Zipkin 提供了可视化服务依赖关系、可视化服务调用时序和获取服务调用数据等强大功能。在本章中，我们也基于 Spring Cloud Sleuth 和 Zipkin 给出了实现自定义跟踪的方法。

下一章将讨论构建微服务过程中另一个重要主题，即如何对微服务进行测试以便确保服务交付的质量。

第 11 章

Spring Test 与服务测试

对于软件中的任何功能，我们都需要进行测试。测试是一门综合性的技术，很多测试的理念和方法都具有通用性，但本书不打算对这些测试的基本概念和实现方式做过多介绍，而是专注于微服务测试这一角度，探讨在微服务架构中开展测试工作与其他功能性测试或非功能性测试在操作过程中的不同点，并给出相应的工程实践。

图 11-1 给出了微服务架构中服务与服务之间最基本的内部结构和交互模式，可以看到在服务内部我们需要关注服务的业务领域模型和数据，而服务之间的交互需要考虑服务协议和远程调用。

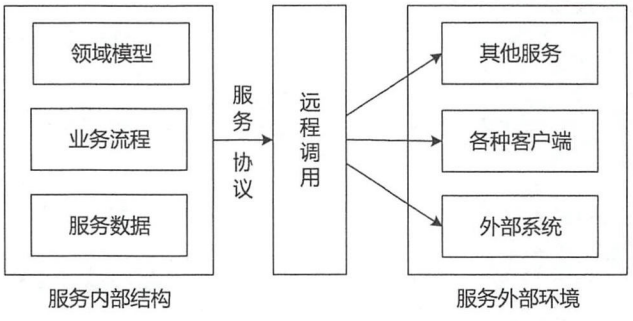


图 11-1　微服务的内部结构和外部交互模式

图 11-1 展示了微服务测试的各个方面，也给我们带来了如下两个基本问题，本章内容将围绕这两个问题做具体展开。

- 微服务测试方法和内容

针对微服务的内部结构和外部交互方式，我们应该采用不同的测试方法和策略。例如，如果关注于具体业务领域中的某一个实体或值对象，那么简单采用单元测试就能满足需求；如果涉及数据访问，则应该借助于集成测试；当不同服务之间进行交互和集成时，测试的关注点就变成如何确保服务定义和协议级别的正确性，这时候就需要采用不同的测试内容。本章将从单元测试、集成测试和端到端测试这 3 种测试方法出发讨论不同内容的测试过程和结果。

- 微服务测试工具和框架

不同的测试方法需要使用不同的测试工具和框架。以 xUnit 为代表的单元测试工具以及各

种 Mock 框架同样适用于微服务测试。同时，因为本书以 Spring Boot 和 Spring Cloud 作为主要的微服务实现技术，也可以采用 Spring Test 作为基础测试框架。Spring Test 集成了 JUnit 并针对数据库访问、Web 等特定场景提供了专门的解决方案。最后，本章还将重点介绍微服务架构中特有的消费者驱动的契约测试框架，消费者驱动的契约测试能够有效应对服务与服务之间交互场景下的端对端测试需求。

11.1 微服务测试的方法

测试存在一些固有的分类，不同类型的测试应对不同的风险。常见的测试类型包括单元测试、集成测试、UI 测试、手工测试等，在具体讨论微服务测试方法之前，先来回顾一下这些测试类型。

11.1.1 单元测试

单元测试（Unit Test）由开发人员实现和使用，测试的对象是小段代码，目的是帮助开发人员确定源代码做了希望它做的事。同时，我们还应该注意到一个单元测试用例只能针对一个类，而且一个类的单元测试必须是完全独立的，不应该与其他代码有任何交互。这也是我们要使用 Mockito、Easymock 等 Mock 框架的原因，当测试过程中某个对象必须和其他对象交互时，可以使用 Mock 对象来保持单元测试的独立性。单元测试可以是自动的，也可以是手动的。通常，我们会在系统的开发阶段手动执行单元测试，但开发和维护单元测试的一个重要目的是为了能够在后续的功能回归中自动化运行从而验证回归测试的效果。

单元测试通常是简单的，因为它的影响范围只在一个类之内，我们只需要根据业务逻辑编写相应的测试用例即可。但是当这个类依赖其他类或组件时，事情就变得没有那么简单。

1. Mock 和 Stub

在介绍如何在单元测试中处理一个类与其他类或组件之间的依赖关系之前，先来明确两个在测试领域比较容易搞混的概念，即 Mock（模拟）和 Stub（打桩）。

首先，要明确一点，Mock 和 Stub 都可以用来对系统（或者将粒度放小为类和组件）进行隔离。在单元测试中，我们通常关注的是测试对象本身的功能和行为，对于测试对象涉及的一些依赖，我们仅仅关注它们与测试对象之间的交互，比如是否调用、何时调用、调用的参数、调用的次数和顺序以及返回的结果或发生的异常等。至于这些被依赖对象如何执行这次调用的具体细节，通常我们并不关注。因此常见的技巧就是使用 Mock 对象或者 Stub 对象来替代真实的依赖对象，模拟真实场景来开展测试工作。

然而，Mock 和 Stub 的区别也非常明显，从类的实现方式上看，Stub 有一个显式的类实现，按照 Stub 类的复用层次可以实现为普通类（被多个测试用例复用）、内部类（被同一个测试用例的多个测试方法复用）乃至内部匿名类（只用于当前测试方法）。对于 Stub 的方法也会有具体的实现，哪怕简单到只有一个简单的 return 语句。而 Mock 则不同，Mock 的实现类通常采用 easymock、jmockMock 等工具包来隐式实现，具体 Mock 方法的行为则通过模拟的方式来实现。通常，对于某一个或一些被测试对象所依赖的测试方法而言，编写 Mock 相对简单，只需要模拟被使用的方法。而 Stub 要复杂一些，需要实现这个类的所有逻辑，即使是不需要关注的方法也至少要给出空实现。

从设计角度来看这两种实现方式，需要引入两个概念，即 interaction-based 和 state-based，读者可参考 Martin Fowler 的一篇文章“Mocks Aren't Stubs”来获取对这两个概念的权威解释。这里只给出结论，Stub 是 state-based，关注的是输入和输出，而 Mock 是 interaction-based，关注的是交互过程。

2. 单元测试的策略

Mock 和 Stub 在单元测试中应用非常广泛，这点对微服务而言同样适用。图 11-2 描绘了在微服务中使用 Mock 和 Stub 的示意图。在现实开发过程中，我们当然也可以使用真实的类或组件来消除测试对象对它们的依赖，所以在图 11-2 中也添加了这种实现方式。

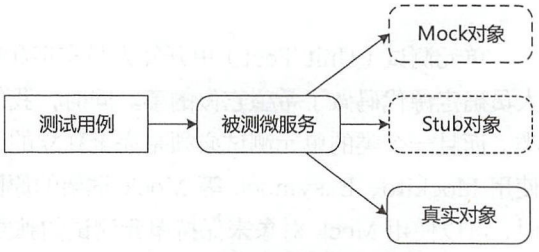


图 11-2 单元测试与依赖关系

在形式上，Mock 是在测试代码中直接 Mock 类和定义 Mock 方法的行为，测试代码和 Mock 代码通常是放在一起的，因此测试代码的逻辑也容易从测试用例的代码上体现出来。而对于 Stub 而言，在测试逻辑比较复杂的场景中，Stub 数量会很多且某些 Stub 需要传入一些如 true、false 之类的标记来制定不同的行为时，测试逻辑的可读性就会下降。

另外，Mock 通常很少考虑复用，每个 mock 对象都是遵循“just enough”原则，一般只适用于当前测试方法。而 Stub 则通常比较方便复用，尤其是一些通用的 Stub。所以 Mock 的优点是构建成本低但维护成本较高，而 Stub 则刚好相反，具体采用何种策略需要追求一定的平衡性。

3. 单元测试的内容

在测试内容上，我们认为服务内部的部分领域模型适合开展单元测试。而复杂业务领域模型和服务数据由于一般都会涉及数据库、外部资源、其他服务等依赖，应该采用的测试方式是集成测试。

11.1.2 集成测试

集成测试（Integration Test）的目的在于测试各个组件间是否能互相配合并正常工作。相较单元测试，集成测试的范围比较灵活。一方面范围可以很小，比方说在一个测试用例中涉及了多个类，就可以认为这是集成测试；另一方面，范围也可以大到整个系统，包括从后台到前端的所有组件。显然，集成测试往往会涉及外部组件，如数据库、硬件、网络等。

在单元测试的基础上，将所有模块按照设计要求组装成子系统或系统，需要进行集成测试。一些类和组件虽然能够单独工作，但并不能保证连接起来之后也能正常运作。一些局部反映不出来的问题，在全局上很可能暴露出来，集成测试的意义就在于此。

1. 集成测试的策略

集成测试的策略包括自顶向下集成、自底向上集成、三明治集成、分层集成等多种具体方式。以最常使用的自底向上集成测试为例，实施过程中一般从具有最少依赖性的底层模块开始，按照由底向上的顺序构造系统并进行集成测试。因为模块是自底向上进行组装的，对于一个给定层次的模块，它的子模块以及子模块的所有下属模块事前都已经完成组装并经过测试，所以可以采用这种迭代和累加式策略完成整个系统的测试。

2. 集成测试的内容

对于微服务而言，集成测试的关注点主要在于需要确保服务内部数据和复杂业务流程的正确性，这里的数据来源一般包括关系型数据库、各种 Nosql 或垂直化搜索引擎等；而复杂业务流程则主要面向多个内部服务和数据访问组件之间的整合。

11.1.3 端到端测试

端到端测试（End to End Testing），也就是通常所说的系统测试（System Testing）。很多时候，我们会认为端到端测试是一种黑盒测试，但从某种意义上来说，端到端测试实质上可以理解为一种“灰盒”测试，一种整合了白盒测试和黑盒测试各自优点的测试方法。

1. 端到端测试的策略

取决于系统中测试的具体内容，端到端测试也存在很多种不同的类型。在微服务架构中，情况可能变得更加复杂。例如，在案例 OrderSystem 中，用户的下单流程需要 Order 服务、Product 服务和 Account 服务之间完成复杂的接口交互。为了完成整个业务流程，端到端测试不得不考虑的问题是如何管理服务与服务之间的数据和状态传递，显然这种端到端测试执行成

本非常高。

如果有一种测试方法能够让开发人员在有限的开发环境中完成服务与服务之间的端到端测试，并且能够提供足够快的测试反馈，无疑能够缓解实施端到端测试过程中面对的各种挑战。目前有一种测试策略为我们提供了解决方案，这就是面向契约的消费者测试（Consumer Driven Contract，CDC），而在 Spring Cloud 中同样提供了专门用于实现面向契约的消费者测试的 Spring Cloud Contract 组件。

2. 端到端测试的内容

在微服务架构中，端到端测试的内容即为各个服务之间的交互接口层，也就是 RESTful 风格下的远程调用层。例如在 OrderSystem 中，通过面向契约的消费者测试就可以实现在不启动各个服务的前提下完成各个服务之间的交互过程的正确性（见图 11-3）。我们将在本章 11.4 节中专门对这种面向微服务架构的端到端测试方法以及相关工具进行详细介绍。

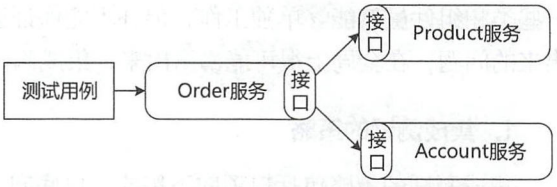


图 11-3　OrderSystem 中的端到端测试内容

11.2　测试 Spring Boot 应用程序

我们知道所有基于 Spring Cloud 开发的微服务本身就是一个 Spring Boot 应用程序，所以对 Spring Boot 应用程序进行测试是微服务测试的基础。本节将初始化 Spring Boot 应用程序的测试环境，并介绍如何在单个服务内部完成单元测试的方法和技巧。

11.2.1　初始化测试环境

Spring Boot 同样提供了针对测试的 spring-boot-starter-test 组件，如下所示。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

在导入 spring-boot-starter-test 依赖之后，我们就可以使用它所提供的各项功能来应对复杂的测试场景。

仅供非商业用途或交流学习使用

1. 测试 Spring 容器可用性

在本书第 2 章中介绍了 Spring Boot 应用程序的各种特点，其中提到在 Bootstrap 类中的 main()入口通过 SpringApplication.run()方法将启动 Spring 容器。如下所示的就是一个典型的 Spring Boot 启动类 OrderApplication，我们可以通过编写测试用例的方式来验证 Spring 容器是否能够正常启动。

```
package com.tianyalan.testing.orders;

@SpringBootApplication
public class OrderApplication {

    public static void main(String[] args) {
        SpringApplication.run(OrderApplication.class, args);
    }
}
```

为了添加测试用例，我们有必要梳理代码的组织结构。图 11-4 所示的就是 order-service 工程中代码的基本目录结构。基于 maven 的默认风格，我们将在 src/test/java 下添加各种测试用例代码。所有的测试用例代码可参考本书提供的案例资源。

图 11-4　order-service 工程代码目录结构

仅供非商业用途或交流学习使用

打开图 11-4 中的 ApplicationContextTests.java 文件，可以得到如下的测试用例代码。该代码中的 testContextLoaded()即为一个有效的测试用例，可以看到该用例简单地对 Spring 中的 ApplicationContext 作了非空验证。

```
package com.tianyalan.testing.orders;

import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.junit4.SpringRunner;

@SpringBootTest
@RunWith(SpringRunner.class)
public class ApplicationContextTests {

    @Autowired
    private ApplicationContext applicationContext;

    @Test
    public void testContextLoaded() throws Throwable {
        Assert.assertNotNull(this.applicationContext);
    }
}
```

执行该测试用例，我们从输出的控制台信息中看到 Spring Boot 应用程序被正常启动，同时测试用例本身也会给出执行成功的提示。

在上述代码中，最重要的就是 ApplicationContextTests 类上的@SpringBootTest 和@RunWith 注解，我们接下来对这两个注解做详细展开。对于 Spring Boot 应用程序而言，这两个注解构成了一套完整的测试方案。

2. @SpringBootTest 注解

因为 SpringBoot 程序的入口是 Bootstrap 类，SpringBoot 专门提供了一个@SpringBootTest 注解来测试 Bootstrap 类。所有配置都会通过 Bootstrap 类去加载，而该注解可以引用 Bootstrap 类的配置。

在上面的例子中，我们直接通过@SpringBootTest 注解所提供的默认功能对作为 Bootstrap 类的 OrderApplication 类进行测试。更常见的做法是在@SpringBootTest 注解中指定该 Bootstrap 类，并设置测试的 Web 环境，如下所示。

仅供非商业用途或交流学习使用


```
@SpringBootTest(classes = OrderApplication.class,
                webEnvironment = SpringBootTest.WebEnvironment.MOCK)
```

@SpringBootTest 注解中的 webEnvironment 可以有 4 个选项，分别是 MOCK、RANDOM_PORT、DEFINED_PORT 和 NONE。

- MOCK

加载 WebApplicationContext 并提供一个 Mock 的 Servlet 环境，内置的 Servlet 容器并没有真实的启动。

- RANDOM_PORT

加载 EmbeddedWebApplicationContext 并提供一个真实的 Servlet 环境，也就是说会启动内置容器，然后使用的是随机端口。

- DEFINED_PORT

这个配置也是通过加载 EmbeddedWebApplicationContext 提供一个真实的 Servlet 环境，但使用的是默认的端口，如果没有配置端口就使用 8080。

- NONE

加载 ApplicationContext 但并不提供任何真实的 Servlet 环境。

在 Spring Boot 中，@SpringBootTest 注解主要用于测试基于自动配置的 ApplicationContext，它允许用户来设置测试上下文中的 Servlet 环境。在多数场景下，一个真实的 Servlet 环境对于测试而言过于重量级，通过 MOCK 环境可以缓解这种环境约束所带来的成本和挑战。我们在下一节会结合 WebEnvironment.MOCK 选项来对服务层中的具体功能进行集成测试。

3. @RunWith 注解与 SpringRunner

@RunWith 注解由 JUnit 框架提供，用于设置测试运行器，例如，可以通过 @RunWith(SpringJUnit4ClassRunner.class)让测试运行于 Spring 测试环境，这里我们指定的是 SpringRunner.class。SpringRunner 实际上就是对 SpringJUnit4ClassRunner 的简化，允许 JUnit 和 Spring TestContext 整合运行，而 Spring TestContext 则是提供了用于测试 Spring 应用程序的各项通用的支持功能。我们在后续的测试用例中将大量使用 SpringRunner。

11.2.2 执行单元测试

本节将通过代码示例回顾如何使用 JUnit 框架执行单元测试的过程和实践，并同时提供验证异常和验证正确性的测试方法。

单元测试的应用场景是针对独立的单个类。如下所示的 OrderNumber 类就是一个非常典型的独立类，该类对 Order 服务中使用的订单编号做了封装，并添加了校验机制。

仅供非商业用途或交流学习使用


```
public class OrderNumber {

    private String orderNumber;

    public OrderNumber(String orderNumber) {
        Assert.notNull(orderNumber, "Order Number must not be null");
        Assert.isTrue(orderNumber.length() == 9, "Order Number must be
            exactly 9 characters");

        this.orderNumber = orderNumber;
    }

    @JsonValue
    public String getOrderNumber() {
        return orderNumber;
    }

    @Override
    public String toString() {
        return orderNumber;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (!(o instanceof OrderNumber))
            return false;

        OrderNumber that = (OrderNumber) o;

        return getOrderNumber() != null ?
            getOrderNumber().equals(that.getOrderNumber())
            : that.getOrderNumber() == null;
    }

    @Override
    public int hashCode() {
        return getOrderNumber() != null ? getOrderNumber().hashCode() : 0;
    }
}
```

仅供非商业用途或交流学习使用

1. 验证异常

在 JUnit 4.7 之前，测试代码中可能抛出的异常类型通常使用类似@Test(expected= IllegalArgumentException) 这一方法。这种方法看上去简单，但存在一个潜在的问题：被标记的测试方法中如果有任何一个操作抛出了 IllegalArgumentException，这个测试用例就会通过。这就意味着抛出异常的地方可能并不是我们期望的那个操作。虽然这种情况可以在设计测试用例时人为地避免，但是还是需要有更好的方法来测试异常抛出。所以在 Junit 4.7 中引入了 ExpectedException Rule 机制，这种机制一方面可以指定期望抛出的异常类型，另一方面还可以指定在抛出异常时所期望得到的异常信息。

在实现上，ExpectedException Rule 机制需要在测试之前使用@Rule 注解来指定一个 ExpectedException，并在测试相应操作之前指定期望的 Exception 类型。引入 ExpectedException 的方法如下所示。

```
@Rule
public ExpectedException exception = ExpectedException.none();
```

引入 ExpectedException 之后，我们就可以验证 OrderNumber 类中各种可能抛出异常信息场景的正确性。例如，验证 OrderNumber 是否为空的测试代码如下所示。根据 OrderNumber 类中的规则定义，一旦构造函数传入的字符串为空，就会触发 Assert.notNull 断言，而该断言会抛出一个带有描述信息的 IllegalArgumentException 异常。我们可以捕捉该异常以验证该断言是否已经被成功触发。

```
@Rule
public ExpectedException exception = ExpectedException.none();

@Test
public void testOrderNumberIsNull() throws
    Exception {

    this.exception.expect(IllegalArgumentException.class);
    this.exception.expectMessage("Order Number must not be null");

    new OrderNumber(null);
}
```

同时，我们也看到 OrderNumber 类中的第二个断言，即构造函数传入的字符串长度必须为 9 个，否则将触发 Assert.isTrue 断言，该断言同样会抛出一个 IllegalArgumentException 异常。该测试用例的实现代码如下所示，分别演示了传入字符串长度多于 9 个和少于 9 个的两种场景。

仅供非商业用途或交流学习使用


```
@Test
public void testOrderNumberIsMoreThan9Chars() throws Exception {

    this.exception.expect(IllegalArgumentException.class);
    this.exception.expectMessage("Order Number must be exactly 9
        characters");

    new OrderNumber("1234567890");
}

@Test
public void testOrderNumberIsLessThan9Chars() throws Exception {

    this.exception.expect(IllegalArgumentException.class);
    this.exception.expectMessage("Order Number must be exactly 9
        characters");

    new OrderNumber("12345678");
}
```

2. 验证正确性

对于单元测试而言，测试异常是一个维度，更多的时候测试用例的目的是期望获取正确的输出结果。例如，针对 OrderNumber 长度问题，可以使用如下测试用例用来验证构造函数中传入字符串满足验证规则的正常业务场景。

```
private static final String ORDER_NUMBER = "123456789";

@Test
public void testOrderNumberIsExactly9Chars() throws Exception {

    OrderNumber OrderNumber = new OrderNumber(ORDER_NUMBER);

    assertThat(OrderNumber.toString()).isEqualTo(ORDER_NUMBER);
}
```

同样，也可以使用如下测试用例用来验证 OrderNumber 类中被覆写的 equals()和 hashCode()方法的正确性。

```
@Test
public void testEqualsAndHashCodeOnOrderNumber() throws Exception {

    OrderNumber OrderNumber1 = new OrderNumber(ORDER_NUMBER);
    OrderNumber OrderNumber2 = new OrderNumber(ORDER_NUMBER);
    OrderNumber OrderNumber3 = new OrderNumber("111111111");
```

仅供非商业用途或交流学习使用


```
        assertThat(OrderNumber1.hashCode()).isEqualTo(
            OrderNumber2.hashCode());

        assertThat(OrderNumber1).isEqualTo(OrderNumber1)
            .isEqualTo(OrderNumber2).isNotEqualTo(OrderNumber3);
    }
```

以上测试用例的执行效果如图 11-5 所示。这些单元测试用例演示了最基本的测试方式，后续的各种测试机制都会在此基础上进行扩展和演化。

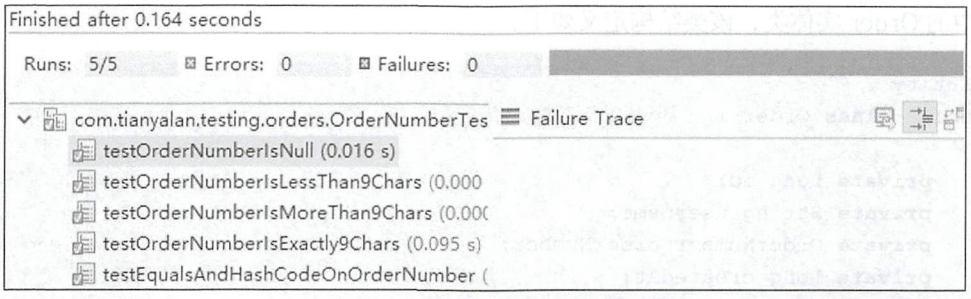


图 11-5 单元测试执行效果图

11.3 使用 Mock 和注解实施集成测试

本节将关注微服务中开展集成测试的方法和工具。在单个微服务内部涉及集成测试的维度有很多，包括数据访问、服务构建和服务集成等。同时，基于常见的代码组织结构，集成测试也体现为一种层次关系，即需要测试从 Repository 层到 Service 层再到 Controller 层的完整业务链路。图 11-6 展示了这种测试层次关系，并给出了各个层次中会用到的主要测试实现方法。

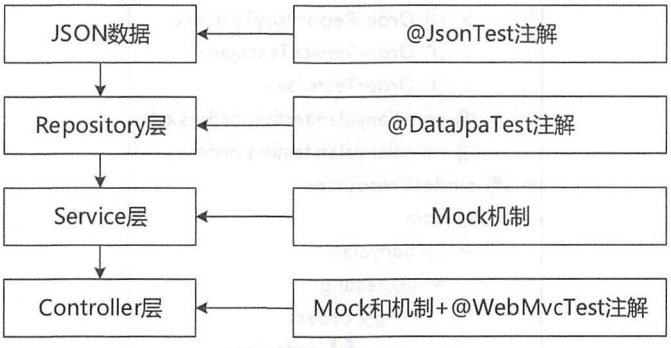


图 11-6 集成测试的层次和实现方式

仅供非商业用途或交流学习使用

11.3.1 使用@JsonTest 注解测试 JSON 数据

为了确保服务的正确性，首先需要确保数据的正确性。在微服务架构中，我们使用 RESTful 风格的 HTTP 协议传输数据，通常数据的序列化/反序列化方式采用的方案是 JSON。作为整个集成测试的最底层测试对象，可以使用@JsonTest 注解测试 JSON 数据，该注解能够完成 JSON 数据文件和内存对象之间的自动映射和转换。

为了演示@JsonTest 注解所提供的功能，我们先来定义一个实体类。这里选择使用 Order 服务中的 Order 实体类，该类结构定义如下。

```
@Entity
public class Order {

    private Long id;
    private String username;
    private OrderNumber orderNumber;
    private Long createdAt;
    private Long lastModified;
}
```

然后准备一个 JSON 数据文件 order.json，并放在 src/test/resources 目录中，请注意该数据文件的文件路径与 classpath 有关。图 11-7 展示了 order.json 文件的存放位置，可以看到它所处的文件路径需要与 com.tianyalan.testing.orders 这一 classpath 保持一致。

图 11-7 JSON 文件路径示意图

仅供非商业用途或交流学习使用

图 11-7 中 order.json 文件的内容如下所示，各个字段名称同样需要与 Order 类中的属性名称保持一致。

```
{
    "createdAt": 1522806163,
    "lastModified": 1522899763,
    "id": 1,
    "username": "tianyalan",
    "orderNumber": "123456789"
}
```

现在来编写测试用例，整个测试用例需要同时考虑序列化和反序列化过程的正确性。这里使用 JacksonTester 工具类并定义了 testSerializeJson()和 testDeserializeJson()这两个测试用例用来完成测试，具体代码如下。

```
@RunWith(SpringRunner.class)
@JsonTest
public class OrderTests {

    private Order order;

    @Autowired
    private JacksonTester<Order> orderJson;

    @Before
    public void setUp() throws Exception {
        Order order = new Order("tianyalan", "123456789");
        order.setId(1L);
        order.setCreatedAt(12345L);
        order.setLastModified(12346L);

        this.order = order;
    }

    @Test
    public void testSerializeJson() throws Exception {
        assertThat(this.orderJson.write(order))
           .isEqualTo("order.json");

        assertThat(this.orderJson.write(order))
           .isEqualToJson("order.json");

        assertThat(this.orderJson.write(order))
           .hasJsonPathStringValue("@.username");
```

```
        assertThat(this.orderJson.write(order))
            .extractingJsonPathStringValue("@.username")
            .isEqualTo("tianyalan");

        assertThat(this.orderJson.write(order))
            .extractingJsonPathStringValue("@.orderNumber")
            .isEqualTo("123456789");
    }

    @Test
    public void testDeserializeJson() throws Exception {
        String content = "{\"username\": \"tianyalan\",
            \"orderNumber\": \"123456789\"}";

        assertThat(this.orderJson.parse(content))
            .isEqualTo(new Order("tianyalan", "123456789"));

        assertThat(this.orderJson.parseObject(content)
            .getUsername()).isEqualTo("tianyalan");
    }
}
```

上述代码中使用了一个@JsonTest 注解，该注解能够自动配置测试用例中所需的 JacksonTester 工具类，以及该工具类背后需用到的 ObjectMapper 等机制。

11.3.2 使用@DataJpaTest 注解测试 Repository 层

数据需要持久化，本节将从数据持久化的角度出发讨论对 Repository 层进行测试的方法，并引入@DataJpaTest 注解。@DataJpaTest 注解会自动注入各种 Repository 类，并会初始化一个内存数据库及访问该数据库的数据源。本章将使用 h2 作为内存数据库，并通过 Mysql 实现数据持久化，因此需要引入以下 maven 依赖。

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

我们需要准备数据库 DDL 用于初始化数据库表，并提供 DML 脚本完成数据初始化。schema-mysql.sql 和 data-h2.sql 脚本分别充当了 DDL 和 DML 的作用。以 Order 服务为例，在 schema-mysql.sql 中包含了 ORDER 表的创建语句，如下所示。

```
DROP TABLE IF EXISTS ORDERS;
CREATE TABLE ORDERS
(
    id                BIGINT(20) PRIMARY KEY NOT NULL,
    created_at        BIGINT(20),
    last_modified     BIGINT(20),
    order_number      VARCHAR(255),
    username          VARCHAR(255)
);
```

而在 data-h2.sql 则完成插入一条测试用数据的工作，初始化数据如下所示。

```
INSERT INTO ORDERS(ID, USERNAME, ORDER_NUMBER) values (1, 'tianyalan',
    '123456789');
```

接下来就是提供具体的 Repository 接口，OrderRepository 接口如下所示，可以看到这里使用了 Spring Data 抽象中的 PagingAndSortingRepository。

```
public interface OrderRepository extends
    PagingAndSortingRepository<Order, Long> {

    List<Order> findOrdersByUsername(@Param("username") String
        username);

    Order findOrderByOrderNumber(@Param("orderNumber") OrderNumber
        orderNumber);
}
```

针对该 OrderRepository 接口的测试用例如下所示，可以看到这里使用了@DataJpaTest 完成 OrderRepository 的注入。同时，我们还注意到另一个核心测试组件 TestEntityManager，TestEntityManager 的效果相当于不使用真正的 OrderRepository 来完成数据的持久化，从而提供了一种数据与环境之间的隔离机制。

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class OrderRepositoryTest {

    private static final OrderNumber ORDER_NUMBER = new
        OrderNumber("987654321");
```

```
    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private OrderRepository orderRepository;

    @Test
    public void testFindOrdersByUsername() throws Exception {
        this.entityManager.persist(new Order("tianyalan2",
            ORDER_NUMBER));

        List<Order> Order = this.orderRepository
          .findOrdersByUsername("tianyalan2");
        assertThat(Order).size().isEqualTo(1);

        Order actual = Order.get(0);
        assertThat(actual.getOrderNumber()).isEqualTo(ORDER_NUMBER);
        assertThat(actual.getUsername()).isEqualTo("tianyalan2");
    }

    @Test
    public void testFindOrderByOrderNumber() throws Exception {
        this.entityManager.persist(new Order("tianyalan3",
            ORDER_NUMBER));

        Order Order = this.orderRepository
            .findOrderByOrderNumber(ORDER_NUMBER);
        assertThat(Order).isNotNull();
        assertThat(Order.getOrderNumber()).isEqualTo(ORDER_NUMBER);
    }

    @Test
    public void testFindOrderByNonExistedOrderNumber () throws
        Exception {
        this.entityManager.persist(new Order("tianyalan2",
            ORDER_NUMBER));

        Order Order = this.orderRepository
            .findOrderByOrderNumber(new OrderNumber("000000000"));
        assertThat(Order).isNull();
    }
}
```

因为我们已经在数据库中初始化了一条 username 为 tianyalan 的数据，所以以上测试用例的结果显而易见。执行这些测试用例，我们需要关注它们的控制台日志输入，其中的核心日志

如下所示（为了显示做了简化处理），从中不难看出执行各种 SQL 语句的效果。

```
Using dialect: org.hibernate.dialect.H2Dialect
org.hibernate.tool.hbm2ddl.SchemaExport  : HHH000227: Running hbm2ddl
schema export
Hibernate: drop table orders if exists
Hibernate: create table order (id bigint generated by default as identity,
created_at bigint, last_modified bigint, order_number varchar(255),
username varchar(255), primary key (id))
org.hibernate.tool.hbm2ddl.SchemaExport  : HHH000230: Schema export complete
…
o.s.jdbc.datasource.init.ScriptUtils     : Executing SQL script from URL …
order-service/target/test-classes/data-H2.sql]
o.s.jdbc.datasource.init.ScriptUtils     : Executed SQL script from URL …
order-service/target/test-classes/data-H2.sql] in 6 ms.
c.t.t.accounts.AccountRepositoryTest     : Started OrderRepositoryTest in
11.779 seconds
o.s.t.c.transaction.TransactionContext   : Began transaction (1) for test
context [DefaultTestContext@10db82ae testClass = OrderRepositoryTests,
testInstance = com.tianyalan.testing.orders.OrderRepositoryTests@6293abcc,
testMethod = testFindOrdersByUsername@AccountRepositoryTests...
Hibernate: insert into orders (id, username, order_number) values (?, ?, ?)
Hibernate: select orders0_.id as id1_0_, orders0_.username as username_2_0_,
orders0_. order_number as order_numbe3_0_ from orders orders0_ where orders
0_.username=?
o.s.t.c.transaction.TransactionContext   : Rolled back transaction for test
context[DefaultTestContext@10db82ae testClass = OrderRepositoryTests,
testInstance = com.tianyalan.testing.orders.OrderRepositoryTests@6293abcc,
testMethod = testFindOrdersByUsername@AccountRepositoryTests..
…
s.c.a.AnnotationConfigApplicationContext : Closing org.springframework.
context.annotation.AnnotationConfigApplicationContext@6b695b06: startup date
[Wed Apr 18 22:04:14 CST 2018]; root of context hierarchy
j.LocalContainerEntityManagerFactoryBean : Closing JPA EntityManagerFactory
for persistence unit 'default'
org.hibernate.tool.hbm2ddl.SchemaExport  : HHH000227: Running hbm2ddl schema
export
Hibernate: drop table orders if exists
org.hibernate.tool.hbm2ddl.SchemaExport  : HHH000230: Schema export complete
```

如果执行 Account 服务中的 AccountRepositoryTests 测试用例，我们得到的也是类似的输出日志。

11.3.3 使用 Mock 测试 Service 层

在本章的 11.2.1 节中我们已经介绍了@SpringBootTest 注解中的 SpringBootTest.WebEnvironment.MOCK 选项，该选项用于加载 WebApplicationContext 并提供一个 Mock 的 Servlet 环境，内置的 Servlet 容器并没有真正启动。本节就针对 Service 层来演示这种测试方式。

首先我们来看一种简单场景，在 Account 服务中存在如下 AccountService 类，可以看到该类只是简单地通过 AccountRepository 完成数据查询操作。

```
@Service
public class AccountService {

    private final AccountRepository accountRepository;

    @Autowired
    public AccountService(AccountRepository accountRepository) {
        this.accountRepository = accountRepository;
    }

    public Account getAccountByUsername(String username) {
        return accountRepository.findAccountByUsername(username);
    }
}
```

显然，对于以上 AccountService 进行集成测试需要提供的依赖就是 AccountRepository，以下代码演示了如何使用 Mock 机制完成对 AccountRepository 的隔离。我们首先通过@MockBean 注解注入 AccountRepository，然后基于第三方 Mock 框架 mockito 提供的 given/willReturn 机制完成对 AccountRepository 中 findAccountByUsername()方法的 Mock。

```
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.context.junit4.SpringRunner;

import com.tianyalan.testing.accounts.model.Account;
import com.tianyalan.testing.accounts.repository.AccountRepository;
import com.tianyalan.testing.accounts.service.AccountService;

import static org.assertj.core.api.Assertions.assertThat;
```

```
import static org.mockito.BDDMockito.given;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
public class AccountServiceTests {

    private AccountService accountService;

    @MockBean
    private AccountRepository accountRepository;

    @Before
    public void before() {
        accountService = new AccountService(accountRepository);
    }

    @Test
    public void testGetUserOrders() throws Exception {
        given(this.accountRepository
            .findAccountByUsername("tianyalan")).willReturn(
                new Account("tianyalan", "tianmin", "zheng",
                    "tianyalan@gmail.com"));

        Account account = accountService
            .getAccountByUsername("tianyalan");
        assertThat(account.getUsername()).isEqualTo("tianyalan");
        assertThat(account.getEmail())
            .isEqualTo("tianyalan@gmail.com");
    }
}
```

以上集成测试的示例中只包含了对 Repository 层的依赖 Mock，有时候一个 Service 中可能同时包含 Repository 和其他 Service 类或组件，如 Order 服务中的 OrderService 类。OrderService 类代码如下所示，可以看到该类同时依赖 OrderRepository 和 AccountClient。

```
@Service
public class OrderService {

    private final OrderRepository orderRepository;

    private final AccountClient accountClient;

    @Autowired
    public OrderService(OrderRepository orderRepository,
        AccountClient accountClient) {
```

```
        this.orderRepository = orderRepository;
        this.accountClient = accountClient;
    }

    public List<Order> getUserOrders() {
        Account account = accountClient.getAuthenticatedAccount();
        if (account == null)
            return Collections.emptyList();

        List<Order> accounts = orderRepository
            .findOrdersByUsername(user.getUsername());
        return accounts;
    }
}
```

请注意以上代码中的 AccountClient 是在 Order 服务中通过 RestTemplate 访问 Account 服务的远程实现类，其代码如下所示。

```
@Service
public class AccountClient {
    private final String serviceHost;
    private final RestTemplate restTemplate;

    @Autowired
    public AccountClient(RestTemplate restTemplate,
        String serviceHost) {

        this.serviceHost = serviceHost;
        this.restTemplate = restTemplate;
    }

    public Account getAuthenticatedAccount() {
        URI url = URI.create(String.format(
            "http://%s/account/v1/tianyalan", serviceHost));

        RequestEntity<Void> request = RequestEntity.get(url)
            .header(HttpHeaders.CONTENT_TYPE, APPLICATION_JSON_VALUE)
            .build();

        return restTemplate.exchange(request,
            Account.class).getBody();
    }
}
```

OrderService 类实际上不需要关注 AccountClient 中如何实现远程访问的具体过程，因为

对于集成测试而言我们只需要关注调用返回的结果。所以，我们对于 AccountClient 同样将采用 Mock 机制完成隔离。针对 OrderService 的测试用例代码如下所示，可以看到我们采用的是同样的测试方式。

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
public class OrderServiceTests {

    @MockBean
    private AccountClient accountClient;

    @MockBean
    private OrderRepository orderRepository;

    private OrderService orderService;

    @Before
    public void before() {
        orderService = new OrderService(orderRepository,
            accountClient);
    }

    @Test
    public void testGetUserOrders() throws Exception {
        given(this.orderRepository.findOrdersByUsername("tianyalan"))
            .willReturn(Collections.singletonList(new Order("tianyalan",
                new OrderNumber("123456789"))));
        given(this.accountClient
            .getAuthenticatedAccount()).willReturn(new Account(1L,
                "tianyalan", "Tianmin", "Zheng"));

        List<Order> orders = orderService.getUserOrders();
        assertThat(orders).size().isEqualTo(1);
        assertThat(orders.get(0).getUsername()).isEqualTo("tianyalan");
        assertThat(orders.get(0).getOrderNumber()).isEqualTo(new
            OrderNumber("123456789"));
    }
}
```

本节中提供的测试用例演示了在 Service 层中进行集成测试的各种手段，这些手段已经能够满足一般场景的需要。

仅供非商业用途或交流学习使用

11.3.4 使用 Mock 和@WebMvcTest 注解测试 Controller 层

我们再回到 Account 服务来看看如何对 AccountController 进行测试。AccountController 类的功能非常简单，代码如下所示。

```
@RestController
@RequestMapping(path = "/account/v1")
public class AccountController {

    private AccountService accountService;

    @Autowired
    public AccountController(AccountService userService) {
        this.accountService = userService;
    }

    @RequestMapping(path = "/{username}")
    public ResponseEntity<Account> me(@PathVariable String username)
        throws Exception {

        return ResponseEntity.ok().body(accountService
            .getAccountByUsername(username));
    }
}
```

在测试 Controller 类之前，先来介绍一个新的注解@WebMvcTest，该注解将初始化测试 Controller 所必需的 Spring MVC 基础设施。AccountController 类的测试用例如下所示。

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure
    .web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

import com.tianyalan.testing.accounts.controller.AccountController;
import com.tianyalan.testing.accounts.model.Account;
import com.tianyalan.testing.accounts.service.AccountService;

import static org.mockito.BDDMockito.given;
```

仅供非商业用途或交流学习使用

```
import static org.springframework.test.web.servlet
    .request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result
    .MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet.result
    .MockMvcResultMatchers.status;

@RunWith(SpringRunner.class)
@WebMvcTest(AccountController.class)
public class AccountControllerTests {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private AccountService accountService;

    @Test
    public void testGetUsers() throws Exception {
        String content = "{\"username\": \"tianyalan\", \"firstName\":
            \"tianmin\", \"lastName\": \"zheng\", \"email\":
            \"tianyalan@gmail.com\"}";

        given(this.accountService.getAccountByUsername("tianyalan"))
            .willReturn(new Account("tianyalan", "tianmin", "zheng",
                "tianyalan@gmail.com"));

        this.mvc.perform(get("/account/v1/tianyalan").accept(
            MediaType.APPLICATION_JSON)).andExpect(status().isOk())
            .andExpect(content().json(content));
    }
}
```

以上代码的关键是 MockMvc 工具类，对这个工具类我们有必要展开一下。MockMvc 类提供的基础方法如下所示。

- perform

执行一个 RequestBuilder 请求，会自动执行 SpringMVC 流程并映射到相应的 Controller 进行处理。

- get/post/put/delete

声明发送一个 HTTP 请求的方式，根据 URI 模板和 URI 变量值得到一个 HTTP 请求，支持 GET、POST、PUT、DELETE 等 HTTP 方法。

仅供非商业用途或交流学习使用

- param

添加请求参数，发送 JSON 数据时不能使用这种方式，而应该使用@ResponseBody 注解。

- andExpect

添加 ResultMatcher 验证规则，通过对返回的数据进行判断来验证 Controller 执行结果是否正确。

- andDo

添加 ResultHandler 结果处理器。

- andReturn

最后返回相应的 MvcResult，然后执行自定义验证或做异步处理。

执行该测试用例，我们从输出的控制台日志中不难发现整个流程相当于启动了 AccountController 并执行远程访问，而 AccountController 中所用到的 AccountService 则做了 Mock。显然测试 AccountController 的目的在于验证其返回数据的格式和内容，我们先定义了 AccountController 将会返回的 JSON 结果，然后通过 perform、accept 和 andExpect 方法最终模拟 HTTP 请求的整个过程并验证结果的正确性。

11.4 消费者驱动的契约测试

对于任何一个类或组件所暴露的对外接口，我们都可以把它们归为一种契约（Contract），即接口的调用者希望通过接口获取某种约定的价值，然而很多时候这种契约并没有被正式的约定。本节将讨论消费者驱动的契约测试，该测试方法一经提出就在微服务架构中得到很好的应用和推广。

11.4.1 面向契约的端对端测试

消费者驱动的契约测试本质上也是一种端对端测试，但也具备一定的特性。要实施消费者驱动的契约测试首先需要了解它的设计理念和系统方法。

1. 消费者驱动契约测试的理念

在微服务中，当服务没有满足约定，错误就会产生在生产环境或集成测试环节。前面介绍的集成测试虽然能够发现并解决部分因为违反接口约定所带来的错误，但集成测试本身也会存在一些问题。最典型的场景就在于随着服务的不断迭代，接口也会相应的产生变化。如果服务数量较多，这种变化会导致集成测试结果的不稳定。图 11-8 展示了由于接口变化导致集成测试失败的一种场景。

仅供非商业用途或交流学习使用

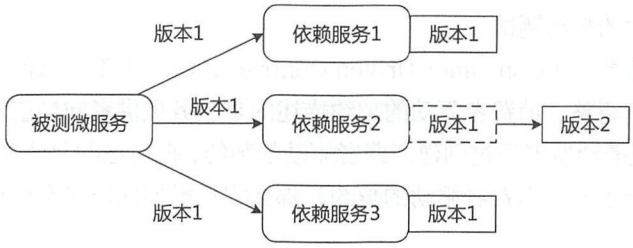


图 11-8　接口版本变化导致集成测试失败的场景

在图 11-8 中，被测微服务依赖服务 1、服务 2 和服务 3 这 3 个不同的服务，在现有情况下，这 3 个服务都开发了第一个版本用于支持这种集成关系。根据业务需要，服务 2 做了一次服务升级，从版本 1 升到了版本 2，我们就会发现集成测试在这个时候可能发生错误，这种错误来自接口的既有约定被打破。

（1）契约的概念

我们可以把接口之间的约定关系用一个词来做抽象：契约。契约由以下几个方面所组成。

- 数据格式

定义其他微服务所期望的数据格式，以及如何将这些数据传递给对方。

- 接口定义

接口所能提供的可用操作。

- 访问协议

能够通过某种协议或步骤完成接口定义中的具体操作。

- 非功能性描述

调用时延或吞吐量等非功能性约束和条件。

对于服务的提供者和消费者而言，存在不同的契约表现形式。服务提供者契约包含了服务提供者所能提供的所有内容，所以一个服务提供者仅包含一种契约，而且这种契约一般会随着版本的演进而不断变化，正如图 11-8 中服务 2 所示的效果一样。

消费者契约则包含对服务进行利用的各个方面，一个服务可以存在一个或多个消费者契约。这种契约只包含某个或某些消费者真正在使用的一部分服务定义，并且根据服务消费者的变更而做响应的调整。图 11-9 展示了服务提供者契约和消费者契约之间的这层关系。

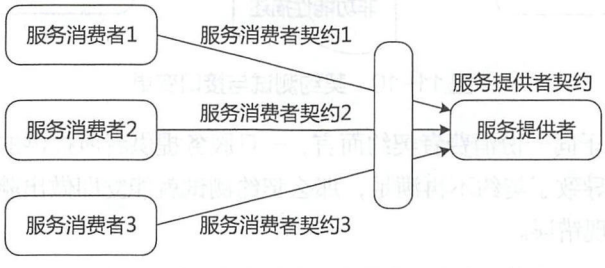


图 11-9　服务提供者契约和消费者契约的关系

仅供非商业用途或交流学习使用

（2）消费者驱动的契约测试

所谓消费者驱动契约（Consumer Driven Contract，CDC）就是从消费者的角度出发驱动消费者协议的制定和调整。消费者驱动的契约描述的是服务提供者向当前所有服务消费者承诺遵守的约束。各个服务消费者通过创建消费者驱动的契约，把自己的具体期望告知服务提供者。如果服务提供者接受了一个消费者驱动的契约，那么它只需要保证已有约束仍能得到满足即可自行修改和发布。

消费者驱动的契约测试是针对微服务接口开展的测试，它能验证服务提供者所提供的契约是否满足消费者的期望。对于一个服务提供者而言，每个消费者会根据与其交互场景和上下文的不同产生不同的契约。当这个服务提供者频繁变更时，就应该保证每个消费者依然能够正确的消费契约。

消费者驱动的契约测试能够提供一定机制验证提供者所提供的服务能否始终满足契约。因为每个消费者拥有自身的消费者契约，所以我们只需要根据消费者契约编写独立的测试用例，并验证这些契约下服务提供者所暴露出来的那一部分接口即可。这些测试用例仅仅关注契约是否满足期望，而不需要深入测试服务内部的行为，所以测试方式相较集成测试而言具有轻量级的优点，可以在很大程度上降低测试成本。

在服务交互过程中，消费者驱动的契约测试能够帮助服务消费者和服务提供者验证接口是否已经发生变化。每当服务提供者所暴露的接口发生变更，契约测试就能检测该接口是否仍然和契约所要求的保持一致，图 11-10 展示了这一过程。

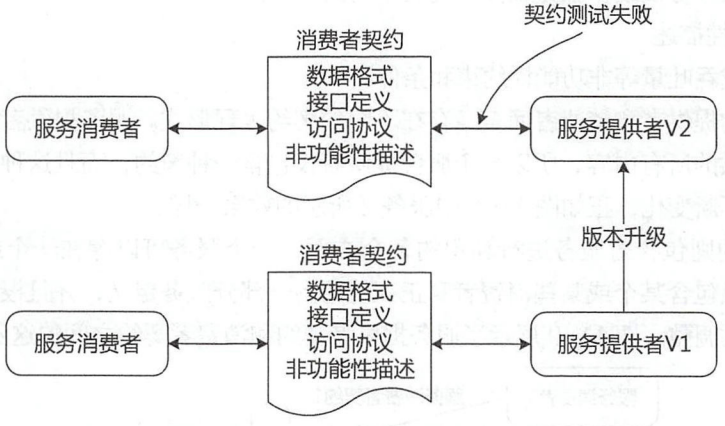


图 11-10　契约测试与接口变更

图 11-10 中，对于同一份消费者契约而言，一旦服务提供者所提供的接口从版本 1 上升到版本 2，如果新版本导致了契约不再满足，那么契约测试就能立即做出验证，从而在开展功能测试之前就能尽早发现错误。

仅供非商业用途或交流学习使用

2. 消费者驱动契约测试的系统方法

消费者驱动契约测试的系统方法包括设计合理的测试过程以及采用合适的测试工具。

（1）消费者驱动契约测试过程

消费者驱动契约测试的目的是为了明确契约的有效性。基于图 11-10 中的思路，消费者驱动契约测试可以采用以下步骤验证契约是否已经发生变更。

- 业务场景提取

根据业务需求选择合适的测试场景。并不是所有的业务场景都需要使用消费者驱动契约测试，往往越容易发生变更的业务场景就越需要进行测试。

- 消费者请求契约化

消费者发送的请求、提供者返回的响应以及关于业务场景的元数据都需要明确记录，并整理成该场景下的契约。

- 模拟消费者发送请求

模拟消费者，向真实的服务提供者发送请求。

- 提供者验证

通过获取请求结果，验证提供者提供的契约是否已经发生变化。

（2）消费者驱动契约测试工具

目前，业界支持消费者驱动契约测试工具常见的有 Pact、Pacto 和 Janus。当然，作为一个完整的微服务套件，Spring Cloud 也提供了 Spring Cloud Contract 作为消费者驱动契约测试的开发框架。这里以 Spring Cloud Contract 为例对消费者驱动契约测试工具的使用方式做详细介绍。关于 Spring Cloud Contract 的更多信息可以参考其官方网站。

11.4.2 实现面向契约的端对端测试

在以 Spring Cloud 作为微服务基础架构的开发环境中，集成 Spring Cloud Contracts 作为消费者驱动契约测试工具是最佳选择。Spring Cloud Contracts 中提供了 Spring Cloud Contracts Verifier 和 Stub Runner 等核心组件，这些组件使得基于 JVM 的消费者驱动契约的开发成为可能。

Spring Cloud Contracts 的设计思想和目的在于生成验收测试和提供快速测试反馈，不需要真正启动所依赖的其他服务就能完成服务契约的正确性验证（见图 11-11）。Contracts Verifier 和 Stub Runner 组件可以确保能够正确 Mock 服务端的接口，并在发生契约变化时提供一种可立即被服务端和消费端发现的方式。

本节中将基于 Order 服务与 Account 服务之间的调用关系来讨论如何基于 Spring Cloud Contracts 实现面向契约的端对端测试。我们对两个服务之间的交互做了简化以便更好地演示

仅供非商业用途或交流学习使用

技术细节。

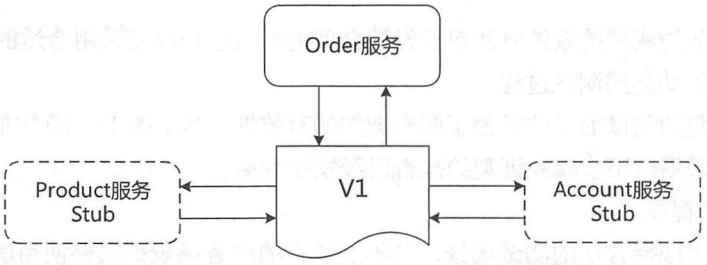


图 11-11　基于 Stub 的契约测试思想

1. 初始化环境

从业务场景上讲，Account 服务相当于服务的提供者，而 Order 服务相当于 Account 服务的消费者。无论是服务的提供者还是消费者，都需要导入以下关于 Spring Cloud Contracts 的 maven 依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-contract-verifier</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-wiremock</artifactId>
    <scope>test</scope>
</dependency>
```

对于 Account 服务而言，我们按照一般的开发流程分别提供了 AccountRepository 和 AccountController 实现，其中，AccountController 的代码结构如下所示（为了演示的简单，去除了 Service 层）。

```
@RestController
public class AccountController {

    @Autowired
```

仅供非商业用途或交流学习使用

```
    private AccountRepository repository;

    @RequestMapping(path = "/v1/accounts")
    public AccountData getCustomers() {
        AccountData data = new AccountData();
        data.setData(repository.findAll());
        return data;
    }
}
```

现在准备工作已经就绪，但 Spring Cloud Contracts 实现面向契约的端对端测试的流程比较复杂（见图 11-12），我们需要对测试流程中的每一步都展开讨论。

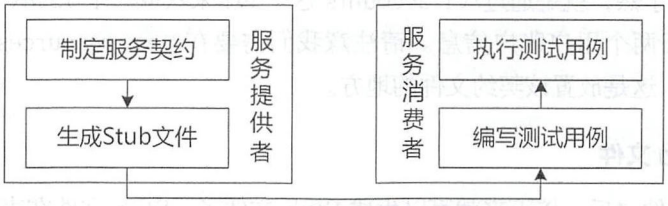


图 11-12　面向契约的端对端测试流程

2. 制定服务契约

引入 Spring Cloud Contracts Verifier 之后就可以使用该组件来定义契约。SpringCloud Contracts Verifier 提供了契约定义语言（Contract Definition Language，CDL）用来定义与契约相关的资源。

Spring Cloud Contracts 中的契约是采用 groovy 的 DSL 进行描述，案例中的 shouldReturnAllAccounts.groovy 契约文件内容如下所示。

```
import org.springframework.cloud.contract.spec.Contract
import org.springframework.http.HttpHeaders
import org.springframework.http.MediaType

Contract.make {
    description "return all accounts"

    request {
        url "/v1/accounts"
        method GET()
    }

    response {
        status 200
        headers {
```

仅供非商业用途或交流学习使用

```
            header(HttpHeaders.CONTENT_TYPE,
                MediaType.APPLICATION_JSON_UTF8_VALUE)
        }
        body("data":[[id: 1L, username: "tianyalan1", firstName:
                    "tianmin", lastName: "zheng", email: "tianyalan@email.com"],
                    [id: 2L, username: "tianyalan2", firstName: "tianmin",
                    lastName: "zheng", email: "tianyalan@email.com"]])
    }
}
```

以上契约文件中包含 3 个部分，即 description、request 和 response。通过 request 定义了请求时的 url 和 method，然后通过 response 约定返回时的 headers 和 body 信息。该契约描述的语义也一目了然，就是通过/v1/accounts 这个 url 来获取一个 JSON 格式的 account 列表，该列表将返回两个用户账户信息。请注意我们需要在 test/resources 目录下新建一个 contracts 文件夹，这是放置该契约文件的地方。

3. 生成 Stub 文件

定义完契约文件之后，接下来就可以生成 Stub 文件了。Stub 文件在表现形式上也是一个 jar 包，这个 jar 包可以被消费者拿来当作一个模拟服务进行启动然后在本地运行测试用例，而不需要服务提供者真正启动服务。

我们首先需要引入 spring-cloud-contract-maven-plugin 插件，spring-cloud-contract-maven-plugin 插件的使用方式如下，该插件在 maven 打包过程中会自动创建 Stub 文件。

```
<build>
  <plugins>
      <plugin>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
      <plugin>
          <groupId>org.springframework.cloud</groupId>
          <artifactId>spring-cloud-contract-maven-plugin
          </artifactId>
          <version>1.1.4.RELEASE</version>
          <extensions>true</extensions>
          <configuration>
              <packageForbaseClasses>
                  com.tianyalan.testing.cdc.accounts
              </packageForbaseClasses>
          </configuration>
       </plugin>
   </plugins>
```

```
    </build>
```

现在我们通过 mvn install-DskipTests=true 命令打包 account-service 工程，除了普通的日志输出，控制台还会生成如下信息（为了显示简单，去掉了文件路径等无用信息）。

```
    [INFO] Copying file shouldReturnAllAccounts.groovy
    [INFO] Converting from Spring Cloud Contract Verifier contracts to WireMock s
tubs mappings
    [INFO] Spring Cloud Contract Verifier contracts directory: …\account-service-
cdc\src\test\resources\contracts
    [INFO] WireMock stubs mappings directory: …\account-service-cdc\target\stubs\
META-INF\com.tianyalan.testing.cdc\account-service-cdc\0.0.1-SNAPSHOT\mappings
    [INFO] Creating new stub […\account-service-cdc\target\stubs\META-INF\com.tia
nyalan.testing.cdc\account-service-cdc\0.0.1-SNAPSHOT\mappings\shouldReturnAllAc
counts.json]
    …
    Installing …\account-service-cdc\target\account-service-cdc-0.0.1-SNAPSHOT-st
ubs.jar to C:\Users\Tianmin.Zheng\.m2\repository\com\tianyalan\testing\cdc\accou
nt-service-cdc\0.0.1-SNAPSHOT\account-service-cdc-0.0.1-SNAPSHOT-stubs.jar
```

根据这些日志信息，我们看到打包过程对 shouldReturnAllAccounts.groovy 契约文件做了处理。打包完成之后，在 target 目录下会生成两个 jar 包，一个是正常的 account-service-cdc-0.0.1-SNAPSHOT.jar 文件，另一个就是新的 Stub 文件。Stub 文件的名称为 account-service-cdc-0.0.1-SNAPSHOT-stubs.jar，打开该文件会发现两个文件夹，一个是 contracts 文件夹，内部存放着 shouldReturnAllAccounts.groovy 契约文件，另一个是 mappings 文件夹，内部存放着 shouldReturnAllAccounts.json 文件，shouldReturnAllAccounts.json 文件是用 JSON 格式对 shouldReturnAllAccounts.groovy 契约文件的一种数据转换。

生成 Stub 文件之后，我们还需要做的事情是通过 install 命令将该 Stub 文件上传到 maven 仓库，以便消费者通过 pom 中定义的 group-id 和 artifact-id 加载该 jar 包。至此，服务提供者的开发工作告一段落。

4. 编写测试用例

现在回到服务消费者端编写测试用例，如下所示的 OrderApplicationTests 类展示了典型的 Spring Cloud Contract 端到端测试用例代码风格。testGetAccounts()测试用例使用 RestTemplate 访问远程 HTTP 端点并验证返回数据的正确性。

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = OrderApplication.class, webEnvironment =
    SpringBootTest.WebEnvironment.RANDOM_PORT)
    @AutoConfigureStubRunner(ids = { "com.tianyalan.testing.cdc:
    account-service-cdc:+:8080" }, workOffline = true)
```

```
public class OrderApplicationTests {

    @Autowired
    private RestTemplate restTemplate;

    @Test
    public void testGetAccounts() {
        ParameterizedTypeReference<AccountData> ptf = new
            ParameterizedTypeReference<AccountData>() {
        };

        ResponseEntity<AccountData> responseEntity =
            restTemplate.exchange("http://localhost:8080/v1/accounts/ ",
            HttpMethod.GET, null, ptf);

        Assert.assertEquals(2,
            responseEntity.getBody().getData().size());
    }
}
```

以上代码中引入了一个新的注解@AutoConfigureStubRunner，通过@AutoConfigure StubRunner 注解来自动注入一个 StubRunner。@AutoConfigureStubRunner 注解包含两个参数，即 ids 和 workOffline。

（1）ids

ids 参数用于定位存放在 maven 仓库中的 stub 包，然后在指定端口启动该 stub 包中的服务。ids 的格式为 groupId:artifactId:version:classifier:port。这里“com.tianyalan.testing.cdc:account-service-cdc:+:8080”表示去 maven 仓库定位上一个步骤中上传的 account-service-cdc-0.0.1-SNAPSHOT-stubs.jar 包并在 8080 端口中启动服务。

（2）workOffline

workOffline=true 是指使用本地 maven 仓库，不要使用线上的版本。显然，作为示例我们可以在本地 maven 完成测试用例的执行。

5. 执行测试用例

执行 OrderApplicationTests，即可在控制台中看到很多日志输出，其中重要的信息如下所示。可以看到在执行过程中 StubRunner 会下载 stub 文件，将该 jar 文件解压到临时目录并启动了内嵌的 Tomcat 监听 8080 端口，然后注册相应的 servlet 并最终运行所有的 stubs。

```
o.s.c.c.s.StubDownloaderBuilderProvider : Will download stubs using Aether
o.s.c.c.stubrunner.AetherStubDownloader : Remote repos not passed but the
switch to work offline was set. Stubs will be used from your local Maven
repository.
```

```
o.s.c.c.stubrunner.AetherStubDownloader : Desired version is [+] - will
try to resolve the latest version
o.s.c.c.stubrunner.AetherStubDownloader : Resolved version is [0.0.1-
SNAPSHOT]
o.s.c.c.stubrunner.AetherStubDownloader : Resolving artifact [com.tianyalan.
testing.cdc: account-service-cdc:jar:stubs:0.0.1-SNAPSHOT] using remote
repositories []
o.s.c.c.stubrunner.AetherStubDownloader : Resolved artifact [com.tianyalan.
testing.cdc: account-service-cdc:jar:stubs:0.0.1-SNAPSHOT] to C:\Users\
Tianmin.Zheng\.m2\repository\com\tianyalan\testing\cdc\account-service-cdc\
0.0.1-SNAPSHOT\account-service-cdc-0.0.1-SNAPSHOT-stubs.jar
o.s.c.c.stubrunner.AetherStubDownloader : Unpacking stub from JAR [URI:
file:/C:/Users/Tianmin.Zheng/.m2/repository/com/tianyalan/testing/cdc/account-
service-cdc/0.0.1-SNAPSHOT/account-service-cdc-0.0.1-SNAPSHOT-stubs.jar]
o.s.c.c.stubrunner.AetherStubDownloader : Unpacked file to [C:\Users\
TIANMI~1.ZHE\AppData\Local\Temp\contracts3346995037700003315]
ationConfigEmbeddedWebApplicationContext : Refreshing org.springframework.
boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@5323999f:
startup date [Thu Apr 19 14:45:57 CST 2018]; root of context hierarchy
f.a.AutowiredAnnotationBeanPostProcessor : JSR-330 'javax.inject.Inject'
annotation found and supported for autowiring
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with port(s):
8080 (http)
o.apache.catalina.core.StandardService : Starting service [Tomcat]
org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache
Tomcat/8.5.23
o.a.c.c.C.[Tomcat-1].[localhost].[/] : Initializing Spring embedded
WebApplicationContext
o.s.web.context.ContextLoader : Root WebApplicationContext:
initialization completed in 512 ms
o.s.b.w.servlet.ServletRegistrationBean : Mapping servlet: 'stub' to [/]
o.s.b.w.servlet.ServletRegistrationBean : Mapping servlet: 'admin' to
[/__admin/*]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080
(http)
o.s.c.contract.stubrunner.StubServer : Started stub server for project
[com.tianyalan.testing.cdc: account-service-cdc:0.0.1-SNAPSHOT:stubs] on
port 8080
o.a.c.c.C.[Tomcat-1].[localhost].[/] : RequestHandlerClass from context
returned com.github.tomakehurst.wiremock.http.AdminRequestHandler.
Normalized mapped under returned 'null'
o.s.c.c.stubrunner.StubRunnerExecutor : All stubs are now running
RunningStubs [namesAndPorts={com.tianyalan.testing.cdc: account-service-
cdc:0.0.1-SNAPSHOT:stubs=8080}]
```

在上面的日志中，我们看到 servlet 生成了/__admin/*映射。该端点在测试用例执行完成

仅供非商业用途或交流学习使用

之后会自动消失，所以可以在 testGetAccounts()方法中打一个断点，然后执行测试用例就可以访问这些自动生成的 HTTP 端点信息。访问 http://localhost:8080/__admin/端点，我们可以获取如下信息，这些信息实际上是根据 shouldReturnAllAccounts 契约生成的用于 stub 的 JSON 数据。Spring Cloud Contract 在内部集成了 WireMock 工具，该工具通过这些 JSON 数据来 Mock 定义的接口。

```
{
    "mappings":[
        {
            "id":"b134a01f-d983-4a05-8889-b1d5aa2e8781",
            "request":{
                "url":"/v1/accounts/",
                "method":"GET"
            },
            "response":{
                "status":200,
                "body":"{"data":[{"id":1,"username":"tianyalan1",
                    "firstName":"tianmin","lastName":"zheng","email":
                    "tianyalan@email.com"},{"id":2,"username":"tianyalan2",
                    "firstName":"tianmin","lastName":"zheng","email":
                    "tianyalan@email.com"}]}",
                "headers":{
                    "Content-Type":"application/json;charset=UTF-8"
                },
                "transformers":[
                    "response-template"
                ]
            },
            "uuid":"b134a01f-d983-4a05-8889-b1d5aa2e8781"
        },
        {
            "id":"02f3f379-ce66-4136-8b35-7b2fd1aafec9",
            "request":{
                "url":"/health",
                "method":"GET"
            },
            "response":{
                "status":200,
                "body":"OK"
            },
            "uuid":"02f3f379-ce66-4136-8b35-7b2fd1aafec9"
        },
        {
            "id":"d5ae77de-dddd-43b3-b1a3-19145ee5582d",
```

仅供非商业用途或交流学习使用


```
            "request":{
                "url":"/ping",
                "method":"GET"
            },
            "response":{
                "status":200,
                "body":"OK"
            },
            "uuid":"d5ae77de-dddd-43b3-b1a3-19145ee5582d"
        }
    ],
    "meta":{
        "total":3
    }
}
```

同时，在执行测试用例的过程中，可以访问 http://localhost:8080/v1/accounts 端点来获取所生成的接口数据，正如我们所预想的一样，返回的数据如下所示。

```
{
    "data":[
        {
            "id":1,
            "username":"tianyalan1",
            "firstName":"tianmin",
            "lastName":"zheng",
            "email":"tianyalan@email.com"
        },
        {
            "id":2,
            "username":"tianyalan2",
            "firstName":"tianmin",
            "lastName":"zheng",
            "email":"tianyalan@email.com"
        }
    ]
}
```

通过整个流程，我们注意到服务提供者是基于消费者的契约来开发接口，而测试用例则是由 Spring Cloud Contract Verifier 根据契约所生成，因此就形成了对契约的一种约束，也就是消费者对服务提供者的约束。如果服务提供者不能满足测试用例则意味着契约已经发生了变化，这正是面向契约的端对端测试的本质所在。该示例的完整代码可以参考本书提供的案例资源。

仅供非商业用途或交流学习使用

11.5 本章小结

在微服务架构中，如何对服务进行测试是一大难题。一方面我们需要验证服务内部功能的正确性，另一方面我们还需要确保服务与服务之间交互过程和结果的正确性。从测试方法上讲，服务测试一般会综合使用单元测试、集成测试和端到端测试等多种方式，本章首先对这些测试方法做了简要介绍。

对于 Spring Boot 应用程序而言，单元测试的开发方式与一般的应用程序一样，但对于集成测试而言则提供了@JsonTest、@DataJpaTest、@WebMvcTest 等多个有用的测试注解，本章对这些常见注解的使用方式做了简要介绍，并结合案例给出了实现方式和代码示例。

本章最后详细讨论了面向契约的端对端测试方法，面向契约测试方法在微服务架构中至关重要，而 Spring Cloud 也提供了专门的 Spring Cloud Contract 组件用于支持面向契约的端对端测试。通过基于契约自动生成 stub 文件和模拟服务启动过程，Spring Cloud Contract 大大简化了端对端测试的流程。本章也基于 Spring Cloud Contract 给出了开发面向契约端对端测试用例的工程实践。

仅供非商业用途或交流学习使用

第 12 章

Docker 与服务部署

当我们把单块系统中的一个实例部署到一台主机或主机集群上，其部署复杂度是 o(n)。若把单块系统拆分成多个微服务之后，其部署复杂度就上升到 o(n2)，由此引出的一个话题就是基础设施的自动化。基础设施的自动化涉及 DevOps 的各个方面，服务部署是其中一个重要的维度。自动化的部署方案能够提高工作效率，降低风险，目前基于容器（Container）的部署机制在业界得到了广泛应用。

Docker 是一款开源的应用容器引擎，我们可以通过 Docker 打包各个微服务以及这些微服务的依赖组件到一个可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。基于 Docker 的容器架构如图 12-1 所示，可以看到这些容器共享操作系统和部分运行时组件库，但各个容器之间是完全隔离的，相互之间不会有任何接口。

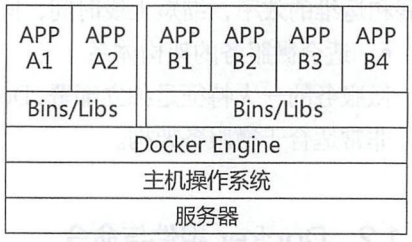


图 12-1 Docker 容器架构图

这种架构带来的好处显而易见，我们可以通过类似搭积木的方式进行软件系统的构建和交付。

在微服务架构中，使用容器技术需要解决以下两个核心问题：

- 如何构建独立的单个服务容器
- 如何组装不同的服务容器构成一个服务体系

本章内容将围绕上述两个核心问题展开讨论，我们将基于 Docker 这一流行的容器技术来完成服务镜像的创建、服务容器的运行以及服务体系的编排，从而实现服务的高效部署。

12.1 Docker 与微服务架构

容器技术的兴起是微服务架构得以高速发展的一大前提，而 Docker 是一种基于 LXC（Linux Container）的应用容器引擎。本节首先简要介绍 Docker 的优势，然后详细分析 Docker 所具备的核心组件与常见命令。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

12.1.1 Docker 的优势

Docker 作为容器为微服务的部署和实施带来了以下转变。

- 轻量级的部署建模方式

Docker 所提供的部署建模方式是轻量级的，也就是说我们在没有使用过 Docker 的前提下把部署工作往 Docker 迁移是一个比较简单的过程。

- 开发和部署职责逻辑分离

通过 Docker 实现的容器管理机制，开发人员关注应用程序，运维人员关注管理容器，从而更好地实现开发和部署的职责分离。

- 快速而高效的开发生命周期

以 Docker 为代表的容器技术以及围绕容器技术展开的基础设施自动化建设能够提升服务部署和运维的效率，缩短上线时间，同时也更加有利于服务的构建和团队的协作。

- 适合微服务的架构体系

微服务的一大特征是独立部署，Docker 恰好就提供了单个容器运行单个应用程序的部署方式，非常适合于微服务架构。

12.1.2 Docker 组件与命令

关于 Docker 的讨论，我们将从两个方面进行切入，一方面介绍 Docker 的核心组件，另一方面则是关注于如何使用 Docker，也就是 Docker 提供的各项命令。

1. Docker 组件

Docker 使用的是 C/S 架构模式，使用远程 API 来管理和创建 Docker 容器。Docker 架构如图 12-2 所示，图中包含了客户端、主机和注册中心 3 个层次，这些层次涵盖 Docker 运行时所涉及的客户端、守护进程、镜像、容器等各个核心组件。我们可以在图 12-2 中看到 Docker 主机分为本地主机和远程主机两类，每个主机都运行着一个守护进程。同时每个主机内部也可能存储着数量不一的镜像和容器，其中守护进程指向镜像，而镜像再指向容器。在介绍如何通过客户端向 Docker 主机发送命令之前，我们有必要对这些 Docker 组件的概念和作用有一个全面的了解。

- 守护进程

守护进程（Daemon）是运作在 Docker 宿主（Host）的后台程序，负责响应来自客户端的请求，也就是各种命令。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

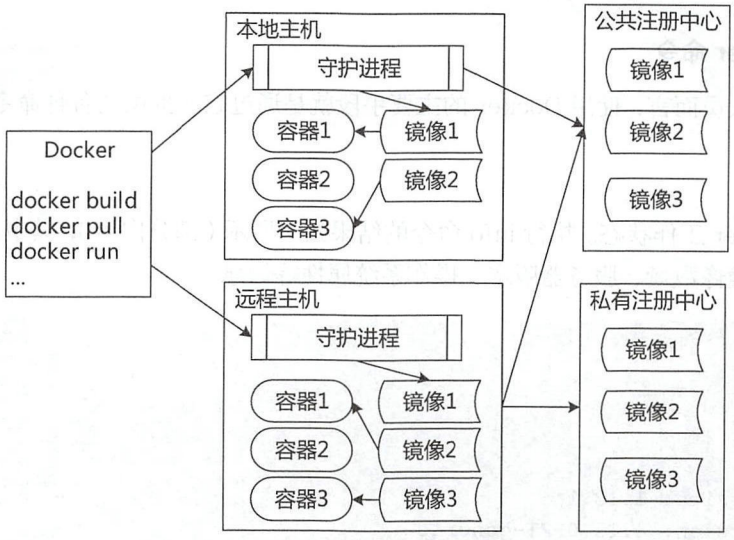


图 12-2　Docker 架构图

- 客户端

客户端（Client）与 Daemon 通信，客户端通过守护进程暴露的各种 API 进行命令的提交。

- 镜像

镜像（Image）相当于源代码的构建环境，由一系列命令组成。在微服务架构中，采用 Spring Cloud 技术体系构建的镜像通常表现为图 12-3 中的形式，包括操作系统、JRE 以及各种基于 Spring Boot 开发的 jar 包。通常，我们构建一个 Docker 镜像并不是完全从头开始，而是在现有的某个镜像的基础上进行扩展（如从 Java 镜像开始构建），Docker 会采用一种分层的文件系统将所有资源全部打包到一起。

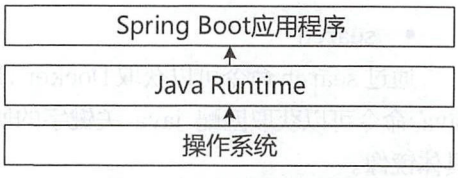


图 12-3　镜像层次结构图

当我们构建一个新的镜像时，只有那些发生变化的部分才会被重新构建。同时，构建结果会被缓存以便为后续的构建过程所复用。

- 容器

容器（Container）是源代码的运行环境，相当于镜像的实例。

- 注册中心

注册中心（Registry）保存已经构建完成的镜像。注册中心分为公共与私有两种，其中，公共注册中心又被称为 Docker Hub，存放着数量庞大的公共镜像，Java EE 领域中的常见工具如 JDK、Tomcat、Mysql 等都有公共镜像；而私有注册中心则面向内部团队，包含着团队所特有的镜像文件。关于公共与私有注册中心的含义，可以类比 maven 的私有仓库和公共仓库的概念。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

2. Docker 命令

对于开发人员而言，使用 Docker 的主要手段就是通过它所提供的各种命令。Docker 的常用命令包括。

- info

查看 Docker 工作状态。执行 info 命令的结果如下所示（部分内容），其中显示了目前运行的容器数量、镜像数量、服务器版本、操作系统属性等信息。

```
Containers: 9
 Running: 2
 Paused: 0
 Stopped: 7
Images: 9
Server Version: 1.13.1
Kernel Version: 4.13.0-21-generic
Operating System: Ubuntu 17.10
OSType: linux
Architecture: x86_64
CPUs: 2
Total Memory: 1.95 GiB
…
```

- search

通过 search 命令可以获取 Docker Hub 中现存的各种镜像信息。例如，执行 docker search java 命令可以获取匹配 java 关键字的所有镜像信息，可以通过镜像名称进一步下载感兴趣的具体镜像。

```
NAME          DESCRIPTION          STARS     OFFICIAL   AUTOMATED
node          Node.js is a JavaScript-basedr...   5141      [OK]
tomcat        Apache Tomcat is an open sourc...  1731       [OK]
java          Java is a concurrent, class-ba...     1643    [OK]
openjdk       OpenJDK is an open-source impl...  841        [OK]
ghost         Ghost is a free and open sourc...     710     [OK]
anapsix/alpine-java Oracle Java 8 (and 7) with GLI...    285     [OK]
jetty         Jetty provides a Web server an...     226     [OK]
…
```

- pull

执行 pull 命令的效果就是从 Docker Hub 中下载最新版的镜像。例如，下载 Java 的最新版本镜像可以使用命令 docker pull java。如果想下载某一个版本的 Java 镜像，则可以添加相应的标签信息，例如，下载 Java7 镜像可以使用命令 docker pull java:7。在首次下载时，该命令

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

会执行镜像的远程下载并保存到本地；如果本地已经存在该镜像，则会输出如下结果，表示镜像已经存在。

```
7: Pulling from library/java
Digest: sha256:
    c0b61b62639124aa838dc755c5a9d57c072f762b71b170281927399a14db4652
Status: Image is up to date for java:7
```

- images

查看镜像列表，执行 docker images 命令的结果如下，分别列出了镜像的所属仓库名称、镜像标签、镜像 ID、镜像创建时间和大小。

```
REPOSITORY      TAG      IMAGE ID       CREATED             SIZE
eureka          0.0.1    2942d4ed6908   53 seconds ago      626MB
hello-world     latest   f2a91732366c   3 months ago        1.85kB
java            7        5dc48a6b75af   13 months ago       584MB
```

- build

使用 Dockerfile 构建镜像，例如，使用存放在/dockerfilepath 路径下的 Dockerfile 并构建名为 eureka:0.0.1 的镜像的命令为 docker build./dockerfilepath/-t eureka:0.0.1。关于如何使用 Dockerfile 构建镜像的各种方式请参考下一节内容。

- rmi

删除镜像，例如，使用 docker rmi eureka:0.0.1 命令表示删除 eureka:0.0.1 这个镜像，而使用 docker rmi-f eureka:0.0.1 命令表示强制删除镜像，这在有些场景下很有用。

- history

查看镜像构建过程，例如，想查看通过 build 命令构建的 eureka 镜像，可以使用 docker history eureka 命令，得到的效果如下所示，可以看到与 build 的过程一致。

```
docker history eureka
IMAGE        CREATED   CREATED BY                                      SIZE
b4e2757a8585  2 hours ago /bin/sh -c #(nop)  ENTRYPOINT ["java" "-ja...  0 B
6c1dca83b5ad  2 hours ago /bin/sh -c #(nop)  EXPOSE 8761/tcp             0 B
3e8a6e4c6d2e  2 hours ago  bin/sh -c #(nop) ADD file:78c663fc5a7abd4... 42 MB
21baeaa17aaa  2 hours ago   /bin/sh -c #(nop)  MAINTAINER tianyalan      0 B
```

- ps

列出容器列表，示例如下。

```
    CONTAINER ID   IMAGE          COMMAND                  CREATED
STATUS              PORTS                              NAMES
    f4cd0e2f1954   eureka:0.0.1  "java -jar /eureka.j…"   About an hour ago
Up About an hour    0.0.0.0:8761->8761/tcp             serene_hypatia
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

- run

拉取镜像，运行容器。该命令可以带有一些参数，如-name 表示命名容器，-d 表示运行的是一个守护式容器，-p 表示映射端口。执行 run 命令的示例如下所示。

```
docker run -d -p 8761:8761 eureka:0.0.1
f4cd0e2f19545c4ac7defe8213bbe9361431386f355d87f96d5daf4de1b155fd
```

- start

启动容器，该命令执行示例如下所示，其中，eb2fccc3878f 为容器的 Id。

```
docker start eb2fccc3878f
eb2fccc3878f
```

- stop/kill

停止/强制停止容器。下面以 stop 命令为例，执行效果如下所示，与 start 一样，eb2fccc3878f 为容器的 Id。

```
docker stop eb2fccc3878f
eb2fccc3878f
```

- rm

删除容器，删除容器之前必须首先停止容器。

以上命令构成了我们使用 Docker 进行镜像构建和容器运行的基础，一般场景下能够满足日常开发的需求。

12.2 使用 Dockerfile 构建服务镜像

本节内容将讨论本章开篇提到的两大核心问题中的第一个，即“如何构建独立的单个服务容器”。在 Docker 中，我们将使用 Dockerfile 构建运行所需的服务镜像。

12.2.1 Dockerfile 命令

在上一节中提到在 Docker 命令里有一项 build 命令使用 Dockerfile 构建镜像，而 Dockerfile 内部包含的就是用于完成镜像构建的一组命令集合，这些命令包括。

- FROM

命令格式：FROM <image>或 FROM <image>:<tag>，指定使用的基础镜像，如 FROM java:8 代表基于 JDK8 开始构建自己的镜像。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

- MAINTAINER

命令格式：MAINTAINER <name>，指定维护者信息，如 MAINTAINER tianyalan。

- RUN

命令格式：RUN <command>或 RUN ["executable", "param1", "param2"]，在当前镜像基础上执行指定命令，并提交为新的镜像。

- CMD

命令格式：CMD ["executable","param1","param2"]，通过该命令可以指定容器启动时执行的命令。因为跟容器启动生命周期有关，所以每个 Dockerfile 只能有一条 CMD 命令。

- EXPOSE

命令格式：EXPOSE <port> [<port>...]，该命令告诉 Docker 服务端容器暴露的端口号供系统交互时使用，如 EXPOSE 8080。

- ENV

命令格式：ENV <key> <value>，该命令用于指定一个环境变量，会被前面的 RUN 命令所使用。

- ENTRYPOINT

命令格式：ENTRYPOINT ["executable", "param1", "param2"]，代表配置容器启动后执行的命令。因为跟容器生命周期有关，所以每个 Dockerfile 只能有一条 ENTRYPOINT 命令。

- ADD

命令格式：ADD <src> <dest>，代表复制指定的 <src> 到容器中的 <dest>，往镜像里添加 jar 包时就需要借助于这条命令。

12.2.2 使用 Dockerfile 命令构建镜像

构建镜像的基础是使用第 12.2.1 节中介绍的各种命令，但构建的方式却有多种。我们可以将 Dockerfile 命令整理成一个文件并手工执行镜像构建，也可以基于 maven 等插件在服务构建的同时自动构建该服务的镜像。

1. 使用 Dockerfile 手工构建镜像

下面给出一个最简单的 Dockerfile 示例，包含如下命令。

```
FROM java:8
MAINTAINER tianyalan
ADD eureka-0.0.1-SNAPSHOT.jar eureka.jar
EXPOSE 8761
ENTRYPOINT ["java", "-jar", "/ eureka.jar"]
```

仅供非商业用途或交流学习使用

以上命令的含义是指我们以 JDK8 为基础镜像开始构建一个镜像，指定该镜像的维护者是 tianyalan，接着往镜像中添加了一个名为 eureka-0.0.1-SNAPSHOT.jar 的 jar 包并重新命名为 eureka.jar，然后暴露服务端口为 8080。最后，通过 ENTRYPOINT 命令在容器启动成功之后运行 java-jar /eureka.jar 命令以启动 eureka.jar。我们是通过 Spring Boot 构建的这个 eureka.jar，所以可以直接使用 java-jar 来启动 jar 包中内置的 Tomcat 服务器，从而启动服务。

手工执行 docker build./docker/ -t eureka:0.0.1 以触发该 Dockerfile 中的命令，执行的过程如下所示，可以看到首先会从公共注册中心中拉取 JDK8，然后依次执行各条命令。这里值得注意的是，每一条命令执行完成之后，都会生成一串类似编号的字符串，这个字符串实际上代表的就是一个新的镜像编号，因为每执行完一条命令，我们都会得到一个新的镜像文件。

```
Step 1/5 : FROM java:8
5040bd298390: Pull complete
fce5728aad85: Pull complete
76610ec20bf5: Pull complete
60170fec2151: Pull complete
e98f73de8f0d: Pull complete
11f7af24ed9c: Pull complete
49e2d6393f32: Pull complete
bb9cdec9c7f3: Pull complete
Digest: sha256:c1ff613e8ba25833d2e1940da0940c3824f03f802c449f3d1815a66b7f8c0e9d
Status: Downloaded newer image for java:8
 ---> 5dc48a6b75af
Step 2/5 : MAINTAINER tianyalan
 ---> Running in 4ccd35b5607a
Removing intermediate container 4ccd35b5607a
 ---> 06576212afeb
Step 3/5 : ADD eureka-server-0.0.1-SNAPSHOT.jar eureka.jar
 ---> 0eaa28a31767
Step 4/5 : EXPOSE 8761
 ---> Running in b02ec5d5e7ee
Removing intermediate container b02ec5d5e7ee
 ---> 9fd54c4e6047
Step 5/5 : ENTRYPOINT ["java","-jar","/eureka.jar"]
 ---> Running in f6699bcdcf2c
Removing intermediate container f6699bcdcf2c
 ---> 2942d4ed6908
Successfully built 2942d4ed6908
Successfully tagged eureka:0.0.1
```

现在我们通过 docker image 命令就会发现已经存在一个 eureka 镜像。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
REPOSITORY      TAG      IMAGE ID         CREATED            SIZE
eureka          0.0.1    2942d4ed6908     53 seconds ago     626MB
```

2. 使用 maven 插件和 Dockerfile 命令构建镜像

使用 maven 插件构建 Docker 镜像的基本方式是在 pom 文件中添加一个 plugin 以改变 jar 包的构建过程。支持 docker 的 maven 插件有很多，下面以 plugin 为例，通过在 pom 文件中添加如下配置就可以实现镜像构建。可以看到该 plugin 中包含了与上一小节中类似的 Dockerfile 命令项，实际上就是相同命令的不同表现形式而已。

```
<plugin>
    <groupId>com.spotify</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <configuration>
        <imageName>${project.name}:${project.version}</imageName>
            <baseImage>java</baseImage>
            <entryPoint>
                ["java", "-jar", "/${project.build.finalName}.jar"]
            </entryPoint>
            <skipDockerBuild>false</skipDockerBuild>
            <resources>
                <resource>
                    <directory>${project.build.directory}</directory>
                    <include>${project.build.finalName}.jar</include>
                </resource>
            </resources>
    </configuration>
</plugin>
```

执行 mvn package docker:build 命令，即可在控制台中看到 Docker 镜像构建的结果，该结果与使用 Dockerfile 手工构建镜像是一致的。

3. 使用 maven 插件读取 Dockerfile 进行构建

当然，如果我们把所有的 Dockerfile 命令整合到一个文件中，就可以使用 Dockerfile 构建镜像，这是 maven 插件最常见的使用方式。我们首先在 Spring Boot 工程中的/src/main/目录下创建一个 docker 文件夹，然后把 Dockerfile 放到该文件夹中，就可以使用如下方式完成镜像的自动化构建，可以看到<dockerDirectory>配置中指向了 Dockerfile 文件的路径。执行 mvn package docker:build 命令同样会生成 docker 镜像，maven 插件对整个过程做了一层封装。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
<plugin>
    <groupId>com.spotify</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <configuration>
        <imageName>${project.name}:${project.version}</imageName>
        <dockerDirectory>
            ${project.basedir}/src/main/docker
        </dockerDirectory>
        <skipDockerBuild>false</skipDockerBuild>
        <resources>
            <resource>
                <directory>${project.build.directory}</directory>
                <include>${project.build.finalName}.jar</include>
            </resource>
        </resources>
    </configuration>
</plugin>
```

以上介绍的 Docker 镜像构建方式都是针对单个微服务，成功构建单个服务镜像是实现多服务编排的基础。

12.3 使用 Docker Compose 编排服务

在 Docker 中运行微服务的方式主要有两种，一种是前面介绍的使用 Dockerfile 构建镜像并启动容器。Dockerfile 方式面向的是单个微服务和单个容器，但微服务架构中往往包含多个微服务。服务数量的增加也就意味着容器数量的增多，逐渐增多的容器数量为容器部署、运行及管理带来了挑战。接下来我们介绍的基于 Docker Compose 编排微服务的方式就是针对这种场景。通过 Docker Compose 可以解决本章开篇提到的另一个核心问题，即“如何组装不同的服务容器构成一个服务体系”。

Docker Compose 面向多个微服务和多个容器，专门用于解决多个容器的部署问题并提高部署方案的可移植性。Docker Compose 是用来定义和运行复杂应用的 Docker 工具，使用 Docker Compose，就可以允许用户在一个模板（YAML 格式）中定义多容器应用，然后通过一条命令来启动整个服务体系。

12.3.1 Docker Compose 组件与命令

在介绍 Docker Compose 的使用方式之前，先来简单了解一下它的组件与命令。

仅供非商业用途或交流学习使用

1. Docker Compose 组件

Docker Compose 的整体架构如图 12-4 所示，从图中我们可以把 Docker Compose 抽象成比较典型的三层结构，包括工程（Project）、服务（Service）以及容器（Contaienr）。

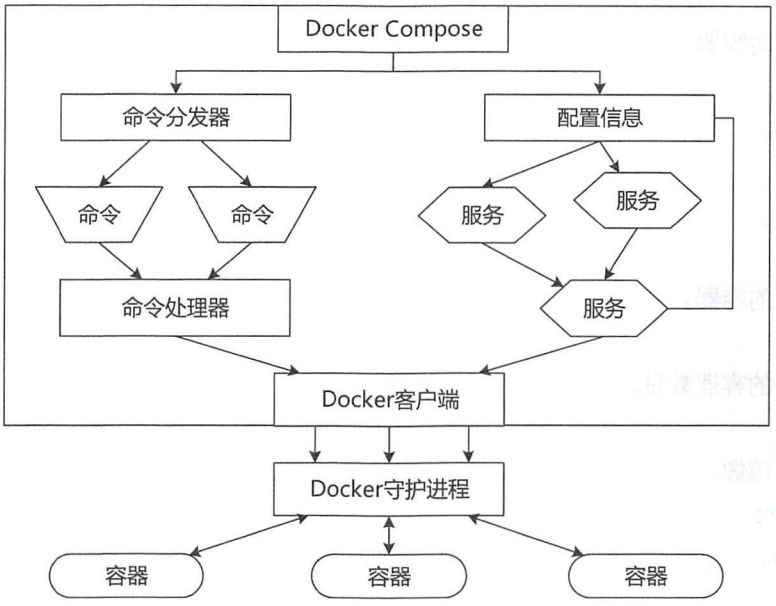


图 12-4 Docker Compose 架构图

（1）工程

这里所说的工程与程序开发过程中的工程概念比较类似，一个工程涵盖了我们需要的所有资源，包括 Docker Compose 运行时所需的所有配置文件及各个服务。Docker Compose 的一个配置文件可以解析为一个项目，即 Docker Compose 通过分析指定配置文件，并得出配置文件所需完成的所有容器管理与部署操作。例如在当前目录下执行后面即将介绍的 docker-compose up-d 命令，相当于使用当前目录下的 docker-compose.yml 配置文件来执行 up 命令。针对该配置文件中的内容，Docker Compose 会将其解析为一个工程。一个工程拥有特定的名称，并且包含一个或多个服务，同时还带有一个 Docker 客户端。

（2）服务

这里的服务是一个抽象的概念，每个服务中定义了容器运行的镜像、参数和依赖关系。一个服务中可以包括多个容器实例。定义一个服务，可以为容器指定镜像或者设定构建镜像用的 Dockerfile，也可以为容器指定与其他容器的连接关系以及端口映射等。

（3）容器

即指 Docker 容器。

仅供非商业用途或交流学习使用

2. Docker Compose 命令

Docker Compose 的常见命令罗列如下，通过英文命名基本都可以明白具体的含义和执行效果。

- up

创建并启动容器。

- kill

杀掉容器。

- ps

显示容器。

- rm

删除停止的容器。

- scale

设置服务的容器数量。

- pull

拉取服务镜像。

- restart

重启容器。

- build

构建或重建容器。

- start

启动容器。

- stop

停止容器。

- down

停止服务并移除容器。

3. docker-compose.yml 命令

想要通过 Docker Compose 编排服务需要提供 docker-compose.yml 配置文件，该配置文件中同样包含了一系列命令用于完成服务的编排，常见的命令如下所示。

- image

指定镜像名称或镜像 Id。如果镜像在本地不存在，Docker Compose 将会尝试拉取这个镜像。

- build

指定 Dockerfile 所在文件夹的路径。Docker Compose 将会利用它自动构建并使用这个

仅供非商业用途或交流学习使用

镜像。

- command

覆盖容器启动后默认执行的命令。

- links

链接到其他服务中的容器。

- ports

暴露端口信息。

- environment

设置环境变量。

12.3.2 使用 Docker Compose

通过编写 docker-compose.yml 配置文件并使用 Docker Compose 命令，可以编排多个服务。本节首先从构建单个服务出发演示 Docker Compose 的基本用法，之后再通过 Docker Compose 编排多个服务。为此我们需要先做好准备工作，包括创建一个 demo 目录并在该目录下添加一个 docker-compose.yml 配置文件，然后在 demo 目录下分别再创建两个目录 eureka 和 config，每个目录下都包含相应的 jar 包和一个 Dockerfile 文件。

1. 使用 Docker Compose 构建单个服务

使用 Docker Compose 构建单个服务的操作非常简单，如果想要单独构建 eureka 服务，只需在 docker-compose.yml 配置文件中添加以下内容即可。

```
version: '2'
services:
  eureka:
    build: ./eureka
    ports:
      - "8761:8761"
```

以上内容中的 ports 命令用于暴露端口信息，使用的是“主机端口:容器端口”的格式，也可以只指定容器端口（主机端口会随机分配）。默认情况下，Docker Compose 会为应用创建一个私有网络，服务的每个容器都会加入到该网络中，每个容器也就可以被该网络中的其他容器所访问。同时，每个容器还能够以服务名称作为主机名被其他容器访问。

执行 docker-compose up 命令，我们会在控制台中得到如下信息，可以看到 Docker Compose 自动创建了一个“demo_default”网络，这里的 demo 就是 docker-compose.yml 配置文件所在的目录。

仅供非商业用途或交流学习使用


```
Creating network "demo_default" with the default driver
Building eureka
Step 1/5 : FROM java:8
---> 5dc8ab75af
Step 2/5 : MAINTAINER tianyalan
 ---> Using cache
 ---> 21baeaa17aaa
Step 3/5 : ADD eureka-server-0.0.1-SNAPSHOT.jar eureka.jar
---> Using cache
 ---> 3e8a6e4c6d2e
Step 4/5 : EXPOSE 8761
---> Using cache
 ---> 6c1dca83b5ad
Step 5/5 : ENTRYPOINT "java","-jar","/eureka.jar"
---> Using cache
 ---> b4e2757a8585
Successfully built 2942d4ed6908
Creating docker2_eureka_1
Attaching to docker2_eureka_1
…
```

等到 eureka 目录下的 Spring Boot 应用程序启动完成之后，访问 http://host:8761/端口即可打开 Eureka 的管理页面。

2. 使用 Docker Compose 编排多个服务

使用 Docker Compose 编排多个服务的示例如下所示，根据需要把具体的服务添加到 docker-compose.yml 配置文件中即可。我们在 eureka 服务的基础上添加了新的 config 服务，并设置了 ports 映射。

```
version: '2'
services:
  eureka:
    build: ./eureka
    ports:
      - "8761:8761"
  config:
    build: ./config
    depends_on:
     - eureka
    ports:
     - "8888:8888"
```

注意这里有一个命令前面章节没有介绍，即 depends_on 命令。depends_on 命令用于指定

仅供非商业用途或交流学习使用

服务依赖，一旦指定了服务依赖，将会优先于自身服务创建并启动依赖。在上面的例子中，config服务依赖eureka服务，通过depends_on命令确保在服务启动顺序上会先启动eureka服务，然后再启动config服务，这也符合Spring Cloud的服务部署要求。

12.3.3 Docker Compose案例分析

回到案例，整个OrderSystem的所有服务如表12-1所示，我们已经在全书的各个章节对这些服务做了详细介绍。

表12-1 OrderSystem服务列表

服务名称	服务描述	服务类型
eureka-server	服务注册中心	基础服务
config-server	分布式配置中心	通用服务
zuul-server	API网关服务器	通用服务
zipkin-server	Zipkin服务器	通用服务
authentication-service	OAuth认证服务	通用服务
order-service	订单服务	业务服务
product-service	商品服务	业务服务
account-service	用户账户服务	业务服务

服务与服务之间势必存在依赖关系，注册中心eureka-server被所有其他服务所依赖，而其他各个服务之间并没有直接的依赖关系。基于以上分析，下面给出案例OrderSystem中所有服务基于Docker Compose的配置信息示例，如下所示。

```
eureka:
  image: "eureka-serve:1.0.0-SNAPSHOT"
  ports:
   - "8761:8761"

config:
  image: "config-server:1.0.0-SNAPSHOT"
  links:
    - eureka
ports:
   - "8888:8888"

zuul:
  image: "zuul-server:1.0.0-SNAPSHOT"
```

仅供非商业用途或交流学习使用


```
    links:
      - eureka
  ports:
    - "5555:5555"

  zipkin:
    image: "zipkin-server:1.0.0-SNAPSHOT"
    links:
      - eureka
  ports:
    - "9411:9411"

  authentication:
    image: " authentication-service:1.0.0-SNAPSHOT"
    links:
      - eureka
  ports:
    - "8080:8080"

  product:
    image: "product-service:1.0.0-SNAPSHOT"
    links:
      - eureka

  account:
    image: "account-service:1.0.0-SNAPSHOT"
    links:
      - eureka

  order:
    image: "order-service:1.0.0-SNAPSHOT"
    links:
      - eureka
```

注意，我们为各个服务都进行了命名，同时，这里还用到了 links 命令，该命令用于链接其他容器服务并通过服务名称进行服务识别。使用 links 命令的同时也和 depends_on 命令一样简单设置了服务依赖关系。

服务之间存在依赖关系的同时也存在层次关系（见图 12-5）。注册中心 eureka-server 属于最基础的服务，显然应该第一个启动。分布式配置中心 config-server、API 网关服务器 zuul-server、Zipkin 服务器 zipkin-server 及 OAuth 认证服务 authentication-service 属于通用服务。而 Order 服务、Product 服务和 Account 服务则属于业务服务。

基于服务层次关系，以上配置内容所生成的网络拓扑如图 12-6 所示。我们知道 Docker 容器之间通过私有网络进行连接，所以 OrderSystem 中的所有服务都位于同一个网络环境中。

仅供非商业用途或交流学习使用

但对于外部系统而言，我们只希望对外暴露基础服务和通用服务，所有的业务服务原则上都应该限制在私有的内网环境中。图 12-6 中的网络拓扑图展示的就是这种服务分层的设计思想。

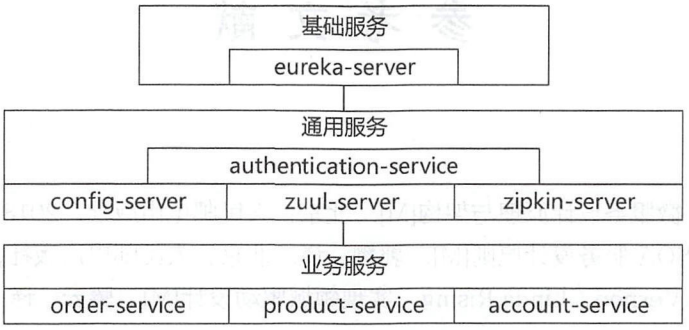


图 12-5　OrderSystem 服务层次关系图

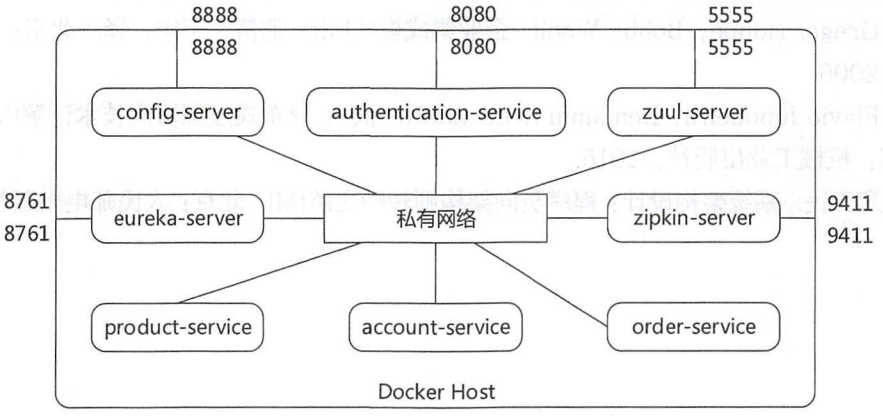


图 12-6　OrderSystem 拓扑结构图

12.4　本章小结

本章作为全书的最后一章讨论了微服务部署相关的话题，并基于 Docker 这一业界主流的容器技术给出了具体的部署方案。

本章首先对 Docker 做了简要介绍，给出了 Docker 组件架构和各种基础命令。然后围绕“如何构建独立的单个服务容器”这一问题展示了使用 Dockerfile 构建 Docker 镜像的 3 种方式。

微服务架构涉及多个服务，所以在单个服务容器构建完成之后，“如何组装不同的服务容器构成一个服务体系”就成为另一个需要解决的问题。使用 Docker Compose 可以实现服务的有效编排。本章最后介绍了 Docker Compose 组件和命令，并结合案例给出了如何使用 Docker Compose 完成服务构建的工程实践。

仅供非商业用途或交流学习使用

参 考 文 献

[1] 郑天民. 微服务设计原理与架构[M]. 北京：人民邮电出版社，2018.

[2] Erl T. SOA 服务设计原则[M]. 郭耀，译. 北京：人民邮电出版社，2009.

[3] Vaughn Vernon，Linda Rising. 实现领域驱动设计[M]. 滕云，译. 北京：电子工业出版社，2014.

[4] 张开涛. 亿级流量网站架构核心技术[M]. 北京：电子工业出版社，2017.

[5] Gregor Hohpe，Bobby Woolf. 企业集成模式[M]. 荆涛，王宇，译. 北京：中国电力出版社，2006.

[6] Flavio Junqueira，Benjamin Reed. ZooKeeper：分布式过程协同技术详解[M]. 谢超，译. 北京：机械工业出版社，2016.

[7] 郑天民. 系统架构设计：程序员向架构师转型之路[M]. 北京：人民邮电出版社，2017.

微服务

架构实战

基于 Spring Boot、Spring Cloud、Docker

作者简介

郑天民，网名天涯兰，日本足利工业大学信息工程学硕士，研究方向为人工智能在大规模调度系统中的应用，在SCI、EI等国际三大索引上发表学术论文4篇，被引用达到50余次。10年软件行业从业经验，在医疗、安防和电商行业都有所涉及，主持和参与过多个大型企业级应用和移动互联网系统的开发和管理工作，前后担任系统分析架构师、部门经理、技术总监等职务。目前就职于一家业界领先的移动医疗互联网公司，负责产品研发与技术团队管理工作。主持过十余个面向研发人员的技术和管理类培训课程，北风网、容大集团等国内知名培训机构合作讲师，对架构设计和技术管理有丰富的经验和深入的理解。著有《系统架构设计：程序员向架构师转型之路》《向技术管理者转型：软件开发人员跨越行业、技术、管理的转型思维与实践》《微服务设计原理与架构》等书籍。

内容简介

本书主要包含微服务架构实现过程中所应具备的技术体系和工程实践，围绕实现微服务架构的基础组件和关键要素，我们将讨论使用Spring Boot、Spring Cloud和Docker等技术体系构建服务治理、负载均衡、服务容错、服务网关、配置中心、事件驱动、服务安全、服务监控、服务测试和服务部署等核心主题，并基于这些核心主题给出具体的案例分析。

免/费/提/供
PPT等教学相关资料

人邮教育
www.ryjiaoyu.com

教材服务热线：010-81055256
反馈／投稿／推荐信箱：315@ptpress.com.cn
人民邮电出版社教育服务与资源下载社区：www.ryjiaoyu.com

封面设计：董志桢

ISBN 978-7-115-49090-2

ISBN 978-7-115-49090-2
定价：59.80 元